NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DESIGN, CONSTRUCTION AND TESTING OF A PROTOTYPE HOLONOMIC AUTONOMOUS VEHICLE

by

Kirk N. Volland

December 2007

Thesis Advisor: Richard Harkins
Second Reader: Peter Crooker

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Design, Construction and Testing of a Prototype Holonomic Autonomous Vehicle		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Kirk N. Volland			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

United States Department of Defense (DoD) autonomous vehicle efforts have concentrated research in areas that support development of unmanned ground and air battlefield vehicles. Little attention has been paid to applying robotics to automate routine tasks. A robotic solution consisting of a prototype holonomic vehicle is proposed to search for, detect, and remove debris that could cause foreign object damage (FOD) to turbine-engine aircraft operated from ships. Holonomic, or omnidirectional, motion was realized by solving the system of equations governing the vehicle's motion atop a plane surface. Translational motion without chassis rotation was achieved through motion control using a single board computer, a pulse width modulation (PWM) and optical isolation circuit, and a low-cost inertial measurement unit (IMU). Obstacle detection and avoidance was realized by constructing a microprocessor-controlled scanning ultrasonic sonar detector head and controller circuit. The sonar detector demonstrated 360º coverage and centimeter resolution. Rudimentary autonomous operation and wireless manual control via a Java graphical user interface (GUI) were achieved in an indoor environment.

14. SUBJECT TERMS Autonomous, Robot, FOD, Foreign Object Damage, Omnidirectional, Holonomic, Odometry, Ultrasonic Sonar, Aviation Safety			**15. NUMBER OF PAGES** 211
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

DESIGN, CONSTRUCTION AND TESTING OF A PROTOTYPE HOLONOMIC AUTONOMOUS VEHICLE

Kirk N. Volland
Lieutenant Commander, United States Navy
B.S. in Technical Communication, University of Washington, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
December 2007

Author: Kirk Volland

Approved by: Richard Harkins
Thesis Advisor

Peter Crooker
Second Reader

James Luscomb
Chairman, Department of Physics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

United States Department of Defense (DoD) autonomous vehicle efforts have concentrated research in areas that support development of unmanned ground and air battlefield vehicles. Little attention has been paid to applying robotics to automate routine tasks. A robotic solution consisting of a prototype holonomic vehicle is proposed to search for, detect, and remove debris that could cause foreign object damage (FOD) to turbine-engine aircraft operated from ships. Holonomic, or omnidirectional, motion was realized by solving the system of equations governing the vehicle's motion atop a plane surface. Translational motion without chassis rotation was achieved through motion control using a single board computer, a pulse width modulation (PWM) and optical isolation circuit, and a low-cost inertial measurement unit (IMU). Obstacle detection and avoidance was realized by constructing a microprocessor-controlled scanning ultrasonic sonar detector head and controller circuit. The sonar detector demonstrated 360º coverage and centimeter resolution. Rudimentary autonomous operation and wireless manual control via a Java graphical user interface (GUI) were achieved in an indoor environment.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The process of creating a novel autonomous vehicle presented me with a formidable challenge that I could never have accepted, let alone succeed at, without the love and support of my family. To my wife, Michelle, I give you my heartfelt thanks. You listened patiently as I explained an endless series of problems related to the project over the course of many months, and often the act of talking it through provided me the course ahead. You allowed me the time to concentrate on completing an autonomous vehicle I could be proud of. Thank you so very much. To my sons, Alec and Kevin, thank you for sacrificing your time with me. To my parents I wish to extend special thanks. Your unique insights were always welcome, and I appreciated your assistance in the preparation of this document.

Professor Harkins, thank you for giving me the opportunity to push forward with the myriad of pioneering ideas I attempted. To James Calusdian, I give my deepest thanks for helping me dive into assembly language programming by sharing your knowledge of PIC microprocessor programming. I wish to thank Sam Barone for his wealth of electronics knowledge and his willingness to help when things became truly strange. I would like to thank George Jaksha for his assistance in crafting numerous parts that comprise the Creature, and I thank Mandy Drury and Donna Shewchuck for their help procuring what Sam and George could not build or unearth from the depths of Spanagel Hall.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

In both civilian industry and the Department of Defense (DoD) robotics continue to grow in importance as the capabilities of computer processors increase and researchers pursue ways to exploit these gains. Presently, robotics comprises a diverse field ranging from stationary industrial machines to vehicles able to traverse planets beyond the reach of any human being to date. Like the National Aviation Space Administration (NASA), the DoD has recognized the utility of autonomous vehicles to complete missions ill-suited to humans due to the mission's hazardous nature, the hostility or remoteness of the environment, its tedious nature, its long duration, or other factors.

A. HISTORIC MILITARY ROBOTIC TASKS AND RESEARCH

Historically, the DoD's efforts in unmanned autonomous ground vehicles have primarily been channeled into robotic devices, such as iRobot's PackBot, to conduct explosive ordnance disposal (EOD) or to provide troops with reconnaissance in urban areas. Figure 1 shows the PackBot in action.

Figure 1. iRobot's PackBot conducting EOD task in Iraq. (From [1])

The obvious desire to spare humans from the need to engage in potentially deadly work such as EOD represents one thrust in DoD robotics. Airborne reconnaissance and weapons employment by unmanned aerial vehicles (UAVs) such as the MQ-1 Predator and its successor, the MQ-9 Reaper, are missions whose characteristics, chiefly long duration and danger to humans, are clearly suited to robotics. Figure 2 shows the U.S. Air Force's newly-operational MQ-9 Reaper. According to General Mosley, the Air Force chief of staff, Reaper's 14 hour endurance is its primary advantage over human-piloted aircraft, not the fact that it removes humans from the battlespace [2].

Figure 2. MQ-9 Reaper UAV. (From [3])

Within the DoD little effort has been devoted to applying robotic solutions to tedious tasks. Rather, DoD and the Defense Advanced Research Projects Agency (DARPA), through its DARPA Grand Challenge and Urban Challenge, have emphasized research into technologies to replace human-operated ground combat vehicles on the battlefield, in support of the Congressional mandate put into law with the National Defense Authorization Act for Fiscal Year 2001 [4]. Figure 3 shows the top three finishers from the 2007 Urban Challenge, a competition designed to promote research into robotic technologies necessary to remove humans from battlefield vehicles.

Figure 3. DARPA Urban Challenge Top Three Finishers. (From [5])

While such efforts are laudable, they neglect the utility of autonomous vehicles for menial, boring, long-duration, routine tasks. Every day men and women in uniform are called upon to complete necessary, albeit time consuming, tasks that waste valuable productive time.

B. PROJECT MOTIVATION

One such task is the ongoing safety and aircraft maintenance requirement to remove debris, or foreign objects, from areas such as flight lines, hangars, hangar bays, and flight decks of U.S. Navy installations and ships. Foreign objects are an amazingly broad class of debris that may include, but is not limited to, the following types of objects:

- aircraft fasteners (e.g. rivets, screws, nuts)
- safety wire
- coins
- tools
- soda cans
- flight gear
- packaging materials (e.g. boxes, wooden splinters from crates)

When a foreign object is ingested by a turbine engine, regardless of whether it is a turbofan, turboshaft, or turboprop engine, the resultant damage, referred to as Foreign Object Damage (FOD) can be costly. If one considers simply the engine replacement cost data available from the Naval Safety Center, then the monetary costs associated with damage to a single F/A-18 E/F Super Hornet jet engine could reach $4.7 million [6]. Engine replacement costs, though, ignore the deadlier potential of FOD. FOD kills. Loss of an engine during a critical phase of flight, such as landing or take off, could result in the complete loss of the aircraft, its aircrew, and the associated loss of the time and money invested in their training. The Naval Safety Center estimates the Navy and Marine Corp spend $90 million annually on FOD-related damage and devote tens of thousands of man-hours to preventing and repairing damage from FOD [7].

Currently, Navy prevention largely consists of "FOD walkdowns," shown in Figure 4, that require anywhere from 30 to 80 people. The FOD walkdown shown in Figure 4 might require 100 people and take 20 minutes; that represents 30 man hours of work. Personnel walk side-by-side and visually scan the deck in front of them to detect FOD and then manually remove it. This task is competed several times each day and is required prior to conducting flight operations. The Navy's current prevention method will become harder to accommodate in the future as the service fields newer ships designed to operate with reduced manning compared with ships presently in service.

C. PROBLEM SOLUTION

An autonomous solution to FOD detection and removal aboard ships could supplement or replace the present manual methods and allow those people to spend their time in more productive ways. Further, a robotic solution would never tire or become distracted, as can occur when humans conduct manual FOD detection and removal.

Figure 4. Typical FOD Walkdown aboard USS CARL VINSON (CVN 70). (From [8])

THIS PAGE INTENTIONALLY LEFT BLANK

II. HOLONOMIC MOTION

A. MOTIVATION

The autonomous vehicle conceived to address the problem of automated FOD detection and removal was dubbed the Creature. Its operating environment was anticipated to include numerous, closely-spaced obstacles. Previous SMART program robots have included conventional four-wheel, skid steered and tracked chassis designs, but these were intended for use outdoors in environments with greater obstacle spacing than the Creature's proposed environment [9,10,11]. To achieve mobility in an environment such as a typical office, it was decided that the design should be capable of rotation in place within its own footprint. Kitagawa, Kobayashi, Beppu, and Terashima, describe the usefulness of a holonomic vehicle in exactly this sort of environment, and note that motion planning for such vehicles is simpler [12]. Robotic researchers have regularly employed holonomic designs in applications where extreme mobility is required, e.g., robot soccer [13]. The researcher decided that the FOD finding robot project offered a good avenue to explore holonomic motion.

B. HOLONOMIC MOTION DEFINITION

In robotics, a holonomic robot possesses an equal or greater number of controllable degrees of freedom (DOF) as the total DOFs of the system [14]. Past literature has also referred to holonomic vehicles as multi-directional or omnidirectional vehicles. Dickerson distinguishes between singular and non-singular multi-directional vehicles. A vehicle with independently steered wheels might be capable of multi-directional motion, but even "a small motion in some directions may require a large motion of the propulsion system," which makes the vehicle singular [15]. In contrast, a non-singular vehicle such as the Creature, "can move a small amount in any direction without a large motion of the wheels" [15]. A holonomic vehicle operating on a plane surface is capable of instantly achieving translational motion in any direction while yawing.

C. DERIVATION OF EQUATION OF MOTION

Kalmár-Nagy, Ganguly, and D'Andrea provide a derivation of the equation of motion for a three-wheeled omnidirectional vehicle as a prelude to their discussion of a less computationally demanding control theory for such a vehicle [16]. Rojas and Förster also provide a concise derivation [17]. We begin by considering an arbitrary Cartesian coordinate frame of reference about the center of mass of the robot. Additionally, assume the center of mass coincides with the geometric center of the robot's circular chassis and is located at the origin, O', as in Figure 5. The positions of the robot's omniwheels about its center of mass are functions of the angle φ.

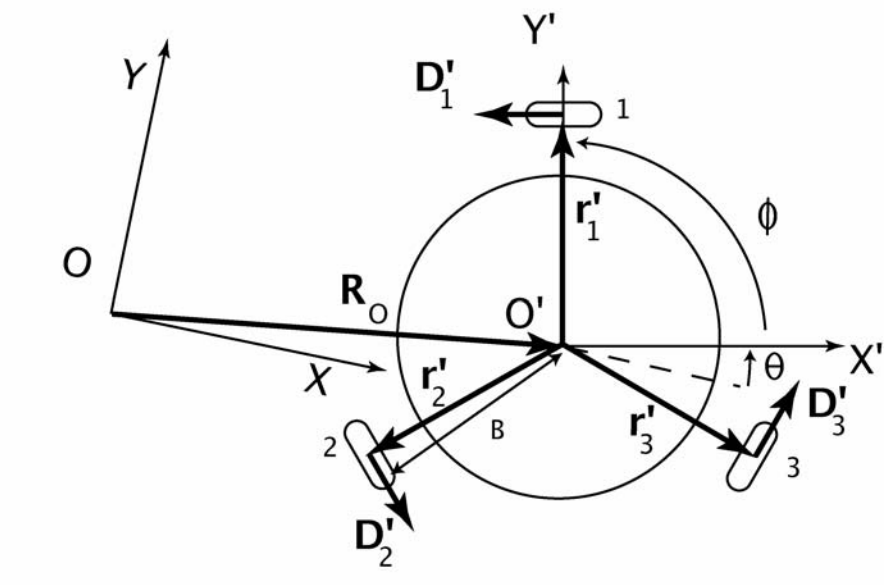

Figure 5. Reference Frames. (After [16])

The vector $\mathbf{r}'_i$, describes the position of the i-th wheel. For a counterclockwise rotation angle, φ, from the X' axis, a rotation matrix R(φ) given by Equation 2.1 can be applied to each wheel position vector to obtain the wheel position in the Cartesian frame of the robot $(X'\ Y')^T$.

$$R(\varphi) = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \tag{2.1}$$

For the first wheel the angle φ is arbitrarily set equal to $\pi/2$, thus aligning $\mathbf{r'}_1$ with the Y' axis. If the wheels are arranged 120º apart and located at distance B from O' then their positions in the robot frame are:

$$\mathbf{r'}_1 = R\left(\frac{\pi}{2}\right) B\begin{bmatrix}1\\0\end{bmatrix} = B\begin{bmatrix}0\\1\end{bmatrix}$$

$$\mathbf{r'}_2 = R\left(\frac{2\pi}{3}\right)\mathbf{r'}_1 = R\left(\frac{2\pi}{3}\right) B\begin{bmatrix}0\\1\end{bmatrix} = B\begin{bmatrix}-\sqrt{3}/2\\-1/2\end{bmatrix}$$

$$\mathbf{r'}_3 = R\left(\frac{4\pi}{3}\right)\mathbf{r'}_1 = R\left(\frac{4\pi}{3}\right) B\begin{bmatrix}0\\1\end{bmatrix} = B\begin{bmatrix}\sqrt{3}/2\\-1/2\end{bmatrix}$$

The drive direction of each wheel is perpendicular to the wheel position vector. The unit vector describing the drive direction, $\mathbf{D'}_i$, of the i-th wheel is obtained by applying the rotation matrix R($\pi/2$) to the position vector of the wheel. For the number one wheel the drive direction is given by Equation 2.2. Equations 2.3 and 2.4 give the drive directions for the number two and three wheels, respectively.

$$\mathbf{D'}_1 = \frac{1}{B}R\left(\frac{\pi}{2}\right)\mathbf{r'}_1 = \frac{1}{B}R\left(\frac{\pi}{2}\right)B\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}-1\\0\end{bmatrix} \tag{2.2}$$

$$\mathbf{D'}_2 = \frac{1}{B}R\left(\frac{\pi}{2}\right)R\left(\frac{2\pi}{3}\right)\mathbf{r'}_1 = \frac{1}{B}R\left(\frac{7\pi}{6}\right)B\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1/2\\-\sqrt{3}/2\end{bmatrix} \tag{2.3}$$

$$\mathbf{D'}_3 = \frac{1}{B}R\left(\frac{\pi}{2}\right)R\left(\frac{4\pi}{3}\right)\mathbf{r'}_1 = \frac{1}{B}R\left(\frac{11\pi}{6}\right)B\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1/2\\\sqrt{3}/2\end{bmatrix} \tag{2.4}$$

The position of the robot's center of mass at O' is related to the Newtonian Earth frame of reference by the vector, $\mathbf{R}_o$. Additionally, the two frames may be rotated by an angle θ with respect to each other, so a rotation matrix, R(θ), can be applied to relate any position vector in the robot's frame to the Earth Frame, $(X\ Y)^T$.

Figure 6 is a simplified view showing the relationship between the positions of the number one wheel in each frame. Let the vector $\mathbf{r}_i$ be the position of the i-th wheel in the Earth frame. Then the position of the wheel, $\mathbf{r}_i$, is the sum of the vector $\mathbf{R}_O$ from the Earth frame origin, O, to the center of mass plus the vector $\mathbf{r}'_i$ rotated to account for the rotation between the frames.

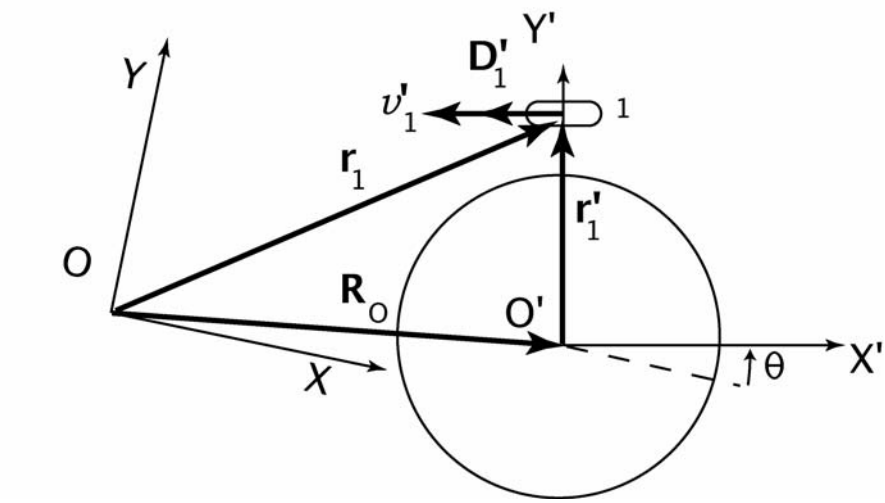

Figure 6. Wheel Position in Earth and Robot Reference Frames.

Equation 2.5 gives the position vector in the Earth frame as a function of the vector from the origin, O, to O' and the rotated vector from the robot frame.

$$\mathbf{r_i} = \mathbf{R}_O + R(\theta)\mathbf{r}_i' \tag{2.5}$$

$$\mathbf{r_1} = \mathbf{R}_O + R(\theta)\mathbf{r}_1' = \mathbf{R}_O + R(\theta)B\begin{bmatrix}0\\1\end{bmatrix}$$

$$\mathbf{r_2} = \mathbf{R}_O + R(\theta)\mathbf{r}_2' = \mathbf{R}_O + R(\theta)R(2\pi/3)\mathbf{r}_1' = \mathbf{R}_O + R(\theta)B\begin{bmatrix}-\sqrt{3}/2\\-1/2\end{bmatrix}$$

$$\mathbf{r_3} = \mathbf{R}_O + R(\theta)\mathbf{r}_3' = \mathbf{R}_O + R(\theta)R(4\pi/3)\mathbf{r}_1' = \mathbf{R}_O + R(\theta)B\begin{bmatrix}\sqrt{3}/2\\-1/2\end{bmatrix}$$

The velocity at the wheel location, $\mathbf{v}_i$, is obtained in the same manner by taking the time derivative in Equation 2.6. Since the vectors $\mathbf{r}'_i$ are constant with time in the robot's frame, their time derivatives are zero, and the velocities in the Earth frame are given by:

$$\mathbf{v}_i = \frac{d\mathbf{R}_O}{dt} + \frac{dR(\theta)\mathbf{r}_i'}{dt} = \dot{\mathbf{R}}_O + \frac{dR(\theta)}{dt}\mathbf{r}_i' + R(\theta)\frac{d\mathbf{r}_i'}{dt}$$

$$\mathbf{v}_i = \dot{\mathbf{R}}_O + \dot{R}(\theta)\mathbf{r}_i' \tag{2.6}$$

$$\mathbf{v}_1 = \dot{\mathbf{R}}_O + \dot{R}(\theta)\mathbf{r}_1'$$

$$\mathbf{v}_2 = \dot{\mathbf{R}}_O + \dot{R}(\theta)\mathbf{r}_2' = \dot{\mathbf{R}}_O + \dot{R}(\theta)R(2\pi/3)\mathbf{r}_1'$$

$$\mathbf{v}_3 = \dot{\mathbf{R}}_O + \dot{R}(\theta)\mathbf{r}_3' = \dot{\mathbf{R}}_O + \dot{R}(\theta)R(4\pi/3)\mathbf{r}_1'$$

At each wheel, i, the wheel velocity, v'_i, must be aligned with the unit vector $\mathbf{D}'_i$ that defines the drive direction for that wheel. Alternatively, one can think of the wheel velocity arising from the dot product of the velocity at the drive point and the drive direction, since the wheel's axis is fixed and perpendicular to the drive direction.

$$v_i' = \mathbf{v}_i \bullet R(\theta)\mathbf{D}_i' \quad \text{or} \quad v_i' = \begin{bmatrix} \mathbf{v}_{ix} \\ \mathbf{v}_{iy} \end{bmatrix} R(\theta)\mathbf{D}_i'$$

One can substitute the velocity $\mathbf{v}_i$ above into the equation, to derive Equation 2.7.

$$v_i' = \begin{bmatrix} \mathbf{v}_{ix} \\ \mathbf{v}_{iy} \end{bmatrix} R(\theta)\mathbf{D}_i' = \left[\dot{\mathbf{R}}_O + \dot{R}(\theta)\mathbf{r}_i'\right]^T R(\theta)\mathbf{D}_i'$$

$$v_i' = \dot{\mathbf{R}}_O^{\ T} R(\theta)\mathbf{D}_i' + \left(\dot{R}(\theta)\mathbf{r}_i'\right)^T R(\theta)\mathbf{D}_i'$$

$$v_i' = \dot{\mathbf{R}}_O^{\ T} R(\theta)\mathbf{D}_i' + \left(\mathbf{r}_i'\right)^T \dot{R}(\theta)^T R(\theta)\mathbf{D}_i'$$

$$v_i' = \dot{\mathbf{R}}_O^{\ T} R(\theta)\mathbf{D}_i' + B\Omega \tag{2.7}$$

In Equation 2.7 the tangential velocity, BΩ, was substituted for the second term in the last step. One can see the wheel velocity contributes to a linear velocity component that produces translational motion, and a rotational velocity component responsible for rotation about the center of mass. If each wheel has radius b and rotates at an angular speed, ω, the wheel velocities, v_i, can be written as in Equation 2.8.

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = b \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \begin{bmatrix} -\cos\theta & -\sin\theta & B \\ 1/2\cos\theta + \sqrt{3}/2\sin\theta & 1/2\sin\theta - \sqrt{3}/2\cos\theta & B \\ 1/2\cos\theta - \sqrt{3}/2\sin\theta & 1/2\sin\theta + \sqrt{3}/2\cos\theta & B \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{2.8}$$

If each wheel's action against the surface produces a force, $\mathbf{f}_i$, due to friction along the $\mathbf{D}'_i$ direction, then the total force on the robot is $\mathbf{F}_R$, which is obtained by summing the three individual drive forces.

$$\mathbf{F}'_R = \sum_{i=1}^{3} \mathbf{f}_i \mathbf{D}_i'$$

In the Earth frame this can be expressed using the rotation matrix.

$$\mathbf{F}_R = \sum_{i=1}^{3} \mathbf{f}_i R(\theta) \mathbf{D}_i'$$

Conservation of linear momentum gives the equation of motion of the robot, which uses the rotation matrix above to relate forces in the robot frame to the Newtonian Earth frame in Equation 2.9.

$$\mathbf{F}_R = M\mathbf{a}$$
$$\sum_{i=1}^{3} \mathbf{f}_i R(\theta) \mathbf{D}_i' = M \frac{d^2 R_O}{dt^2} \tag{2.9}$$

Conservation of angular momentum requires the time rate of change of angular momentum, **L**, equal the torque , $\vec{\Gamma}$, applied.

$$\frac{d\mathbf{L}}{dt} = \vec{\Gamma}$$

Considering only the forces acting parallel to the x-y plane at distance B and the angular momentum, L, about the robot's Z' axis, one can write the conservation equation. Then one can replace the time rate of change of the angular momentum with the moment of inertia, I, and the acceleration of the robot's chassis rotation, which is equivalent to the second derivative with respect to time of the angular displacement between the two coordinate frames as shown in Equation 2.10.

$$\frac{dL_z}{dt} = B\sum_{i=1}^{3} \mathrm{f}_i$$

$$I\frac{d^2\theta}{dt^2} = B\sum_{i=1}^{3} \mathrm{f}_i \tag{2.10}$$

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXPERIMENTAL DESIGN

The Creature is a three-wheeled omnidirectional, or holonomic, robot capable of semi-autonomous operation. It was conceived as a prototype to address the problem of automated FOD detection and removal as well as to provide a research platform for future SMART Program efforts. Its major components are shown in Figures 7 and 8.

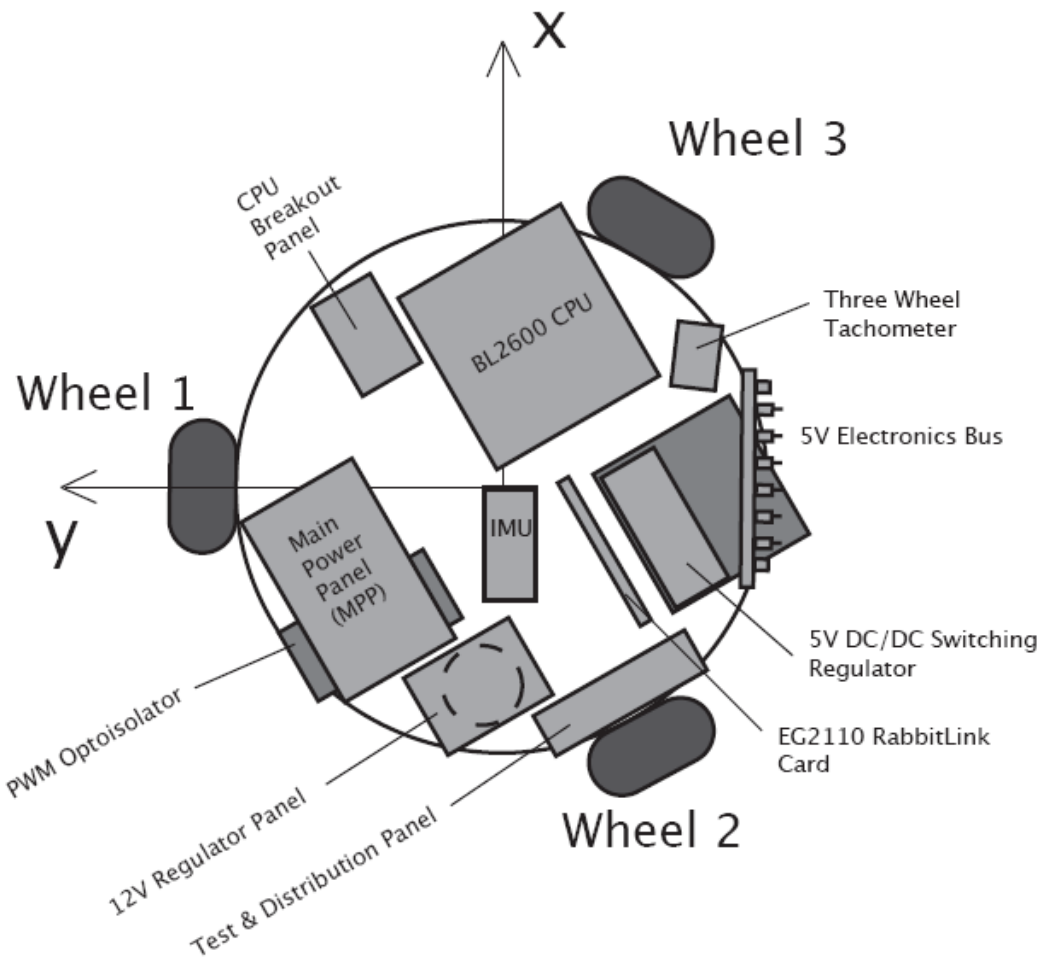

Figure 7. Diagram of Major Components Installed on Lower Level.

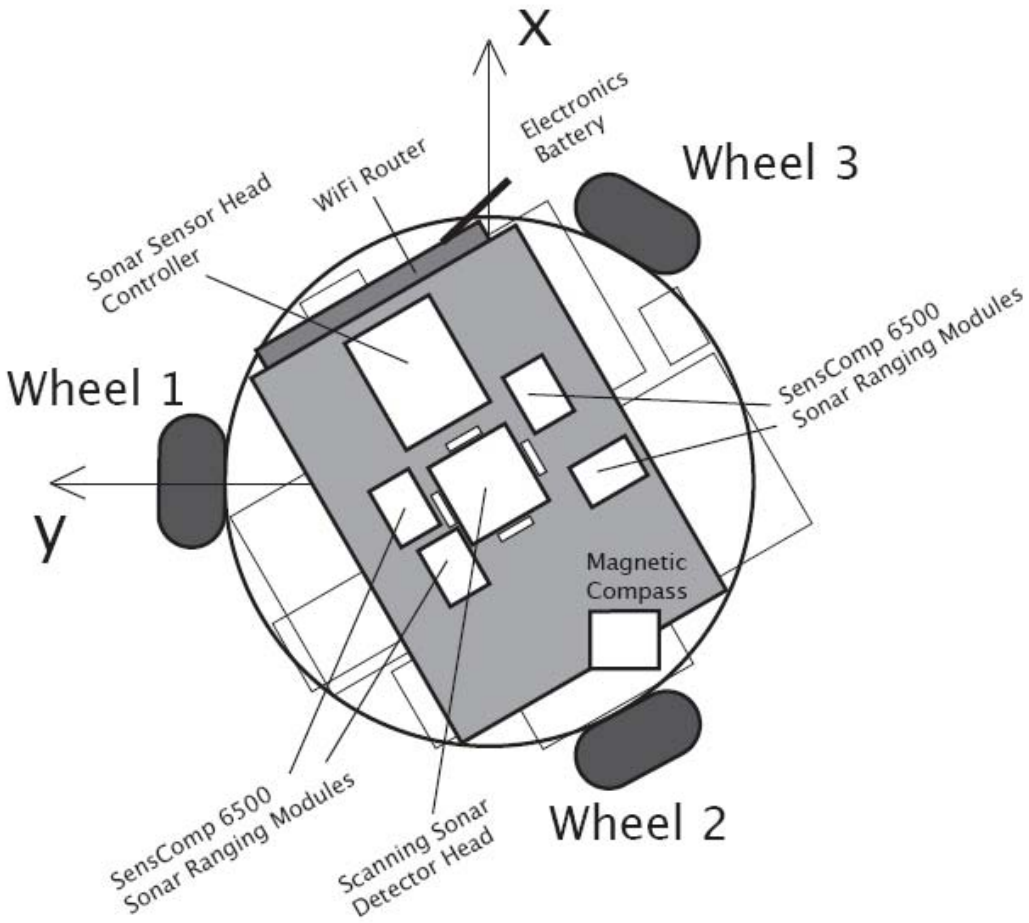

Figure 8. Diagram of Major Components Installed on Upper Level.

A. MECHANICAL CONSTRUCTION

1. Chassis

The chassis design required a platform that was light-weight, strong, and large enough to provide area for mounting components required for the robot's operation, such as batteries, motors, sensors, CPU, etc. A SuperDroid Robots three-wheeled vectoring robot kit consisting of motors, wiring, and chassis was purchased with the intent to use it as the robot's chassis and propulsion. The vendor-supplied equilateral triangular base, measuring eleven inches on a side, was deemed too small because it would not have allowed sufficient space to mount the anticipated equipment or allowed for growth. A larger, low-cost chassis was constructed by reusing a 1/8-inch thick aluminum plate

recovered from a computer's backplane. A seven-inch radius disk with a mass of 838 g was cut from the plate. It was drilled to mount the three motor mounts to the underside of the plate at 120º intervals. The disk also served as a ground plane to isolate the electronics, sensors, CPU, and devices powered by the electronics bus from radio frequency interference (RFI) produced by the three brushed direct current (DC) motors below the chassis.

During construction of the chassis, provision was made to mount three 12 V, batteries on the underside of the chassis disk in the space between each motor mount. Since the batteries were among the most massive components installed, placing them at roughly the same height as the wheel axles was believed to offer improved stability compared to placing them atop the chassis. To increase the robot's moment of inertia about its Z axis, the batteries were placed 11.5 cm from the chassis' geometric center rather than directly under it. Based on the derivation of the equation of motion in Chapter II, one can see that a larger moment of inertia would help maintain directional control in cases where a torque was applied due to imbalances in motor speeds.

The triangular aluminum plate supplied by the vendor was briefly considered for use as a second level for mounting the CPU and sensors. Figure 9 shows the chassis at an early phase of construction with the original triangular top plate. A redesign of the electrical power supplies, described in the Electrical System Design section, required replacing the triangular plate with a larger mounting surface. A Plexiglas square, supported by four-inch-long aluminum standoffs was installed in its place. The square plate provided a larger mounting surface. Sonar sensors were mounted atop it, and the large, flat electronics battery and router are hung beneath it. Four screws attach the top plate to the standoffs, and all devices attached to the top plate feature electrical quick disconnects to allow the operator to remove the top plate in under two minutes for easy access to components mounted atop the chassis.

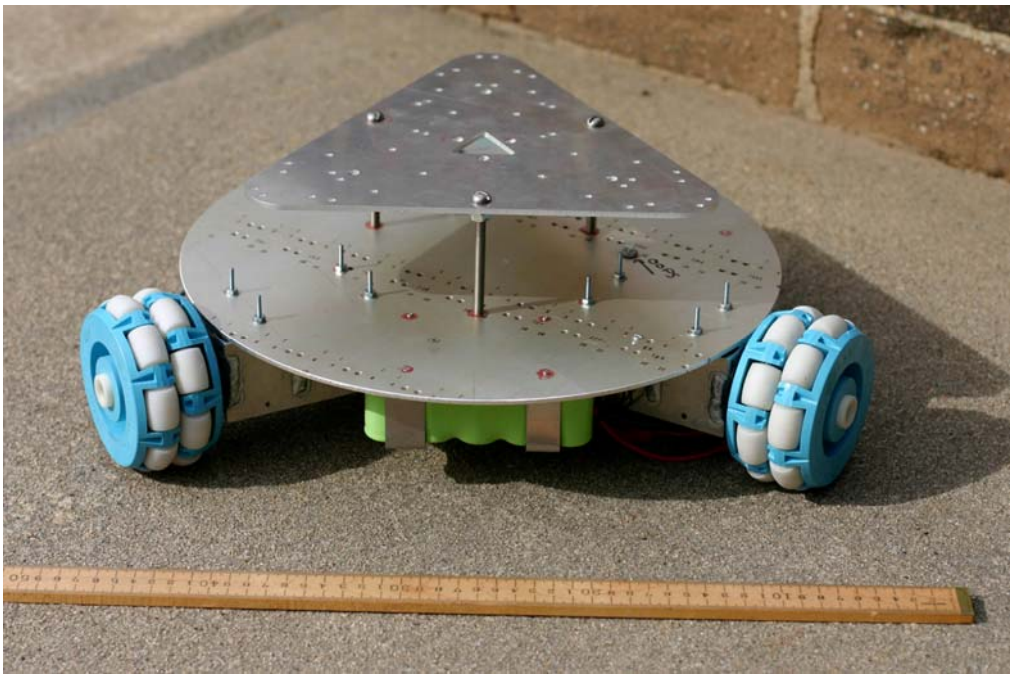

Figure 9. Bare Chassis with Motors Installed.

Figure 10 shows the Creature in a later stage of development. In this photo the first generation electronics bus, visible on the left, is still installed, and the buck switching DC/DC converter has not been installed yet.

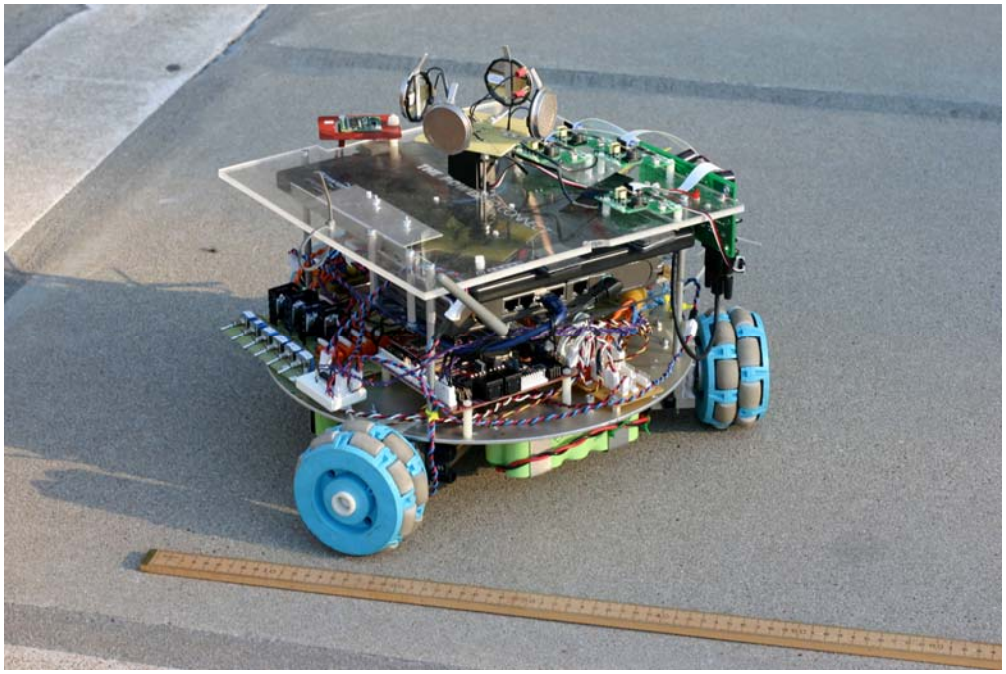

Figure 10. Creature Later Development Configuration.

2. Omniwheels

The heart of the Creature's holonomic mobility is its three Kornylak Transwheels. The plastic wheels have a 4 inch O.D. and two rows of eight free-turning rollers. The wheels' measured diameter was approximately 10 cm, which agrees with the manufacturers' specification. The rollers are oriented 90º to the direction of the motor's axle. The Creature uses the double row model 4000 series, which are rated at 100 pounds per wheel [18]. The Creature's weight during the experimental observations was 20 lbs; thus the load supported by any individual wheel is well within limits.

B. PROPULSION AND CONTROL

1. Motors and Motor Controllers

The Creature is propelled by three 12/24V IG32 ∅32 mm brushed DC gear motors. The motors have a mass of 194 g and a rated speed of 195 rpm. According to the manufacturer's specification sheet, the planetary gear's 1:27 reduction ratio applied to the 6500 rpm no-load motor speed produces an output wheel speed of 240 rpm or 8π radians/s. Rated torque is 1.4 kg cm. Operated at 24 V, the motors draw a maximum of 1.1 A under 350 g cm load [19]. The vendor's motor wiring kit, which includes ferrite rings and shielded hook-up cables, was used to connect the output terminals of the motor controllers to the DC motors' leads. The cables' internal foil shielding is grounded to the chassis to reduce or eliminate unwanted radiated electromagnetic radiation from the DC motor leads, which could act as transmitting antennas.

The motors were installed under the chassis inside the box-like motor mounts supplied with the vendor's vectoring robot kit. The mounts are constructed of welded, 1/8-inch aluminum, and each mount's mass was 205 g. In addition to using the chassis as a ground plane to isolate the CPU and electronics from RFI from the DC motors, each motor was wrapped in a metal screen, forming a small cylindrical Faraday cage around it. Figure 11 shows the underside of the robot. One motor is visible inside its Faraday cage in the photograph. A motor controller is visible on the left. Reducing the length of the motor leads was another RFI reduction measure.

Each motor receives DC electrical power independently via a SuperDroid Robots 3A PWM motor controller. This controller uses an LMD18200 H-bridge motor driver, which is rated at 3 A [20]. The LMD18200 is operated in the sign/magnitude mode, i.e., the controller is supplied two signals, the digital PWM magnitude and the desired motor direction. A lack of PWM signal is interpreted as zero magnitude. Using the sign/magnitude mode provided a finer control of the wheel speeds by allowing the robot's CPU to map its entire analog output range to the desired wheel speed. Previous SMART program robots have operated the same PWM motor controller in a bidirectional magnitude mode, which decreased the resolution of the speed control and resulted in unsteady operation at slow speeds or when stopped [11]. The motor controller's brake signal is not used, and it is held in a low logic state.

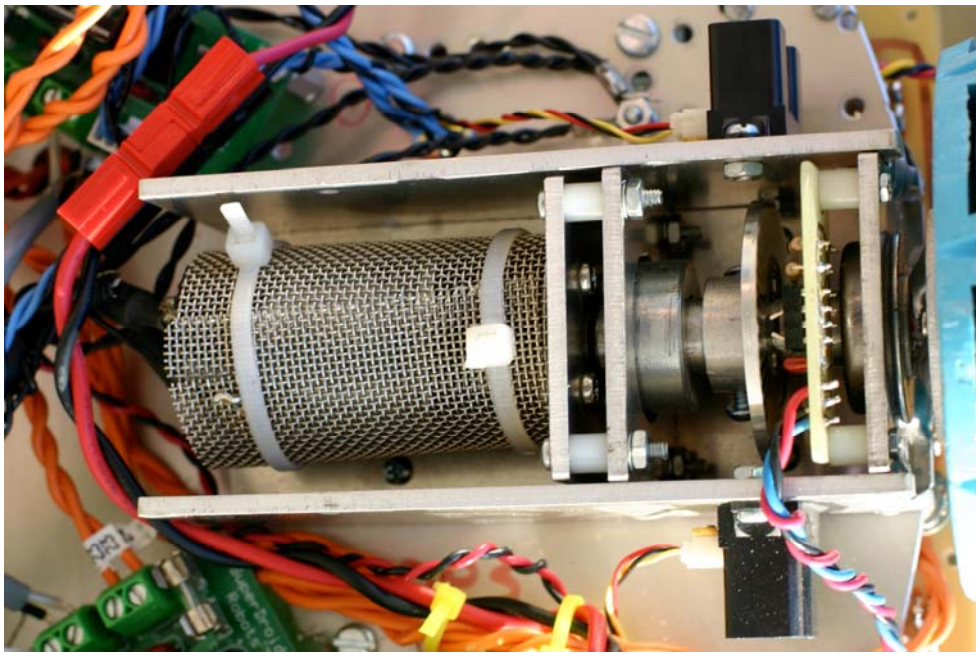

Figure 11. Shielded DC Motor inside Motor Mount.

2. PWM and Optoisolation Circuit

One design goal for the Creature was to distribute the computational load and the CPU's workload. Other autonomous vehicles constructed as part of NPS SMART

program have used the BL2000 single board computer, a 20 MHz processor that offers modest computational speed. It has proven adequate when tasked with running the robot operating program [11]. It was decided that the CPU would control motor speeds with an analog speed signal rather than devote CPU cycles to producing a pulse width modulated (PWM) signal. Three design challenges arose from this. First, the motor controllers described above require a PWM input, so the CPU's analog signal required a circuit able to produce PWM signals from an analog voltage. Second, electrically isolating the CPU on a separate power bus from the motor controllers and DC motors required optical isolation, which in turn required that the CPU's signal be converted to a digital form. Third, the digital direction signals for each motor also needed to be passed from the CPU to the motor controllers, requiring optoisolation for these signals as well.

a. Early Efforts in Pulse Width Modulation of Motor Speed Signals

Intersective pulse width modulation has been implemented previously using the analog speed signal as a reference and comparing it against a regularly varying signal. Miller provides an explanation of a PWM circuit that has been previously used on the Autonomous Ground Vehicle (AGV) SMART program robot. The circuit was based on a LM 555 timer IC [10]. A number of oscillator circuits were explored, including crystal oscillator and LRC tank circuits, in an attempt to improve on the existing PWM circuit.

b. Crystal Oscillator Circuit

A simple 100 kHz crystal oscillator tank circuit using an LM741 opamp was fed to a LM393N comparator. The circuit's schematic is provided below in Figure 12. A significant complication with the crystal oscillator circuit was that it and other sinusoidal oscillator circuits required positive and negative rail voltages. Providing positive and negative rail voltages would have complicated the electronics power supply, so single supply solutions were preferred. Negative triangle wave voltages were not desired, so a diode clamp was used to raise them to a minimum of -0.4 V. Later a 3 kΩ pull-up resistor, not shown, was connected to the PWM signal output to pull-up the output to positive levels.

The circuit showed promising results, and produced a very linear response. In prototype testing the triangle wave of the crystal oscillator and the analog signal were provided to the comparator, and the PWM signal's duty cycle recorded. Then the comparator opamp was provided the reference analog signal and a 100 kHz triangle wave from a function generator. The crystal circuits' data were virtually identical to that obtained with the function generator providing the triangle wave. A linear fit of the crystal circuit's data was obtained; the circuit produced a 0% duty cycle at 0 V, and a 100% duty cycle at 3.730 V. The analog speed signal range corresponded well with the analog output voltage limits of the BL2000, which was the target CPU at the time.

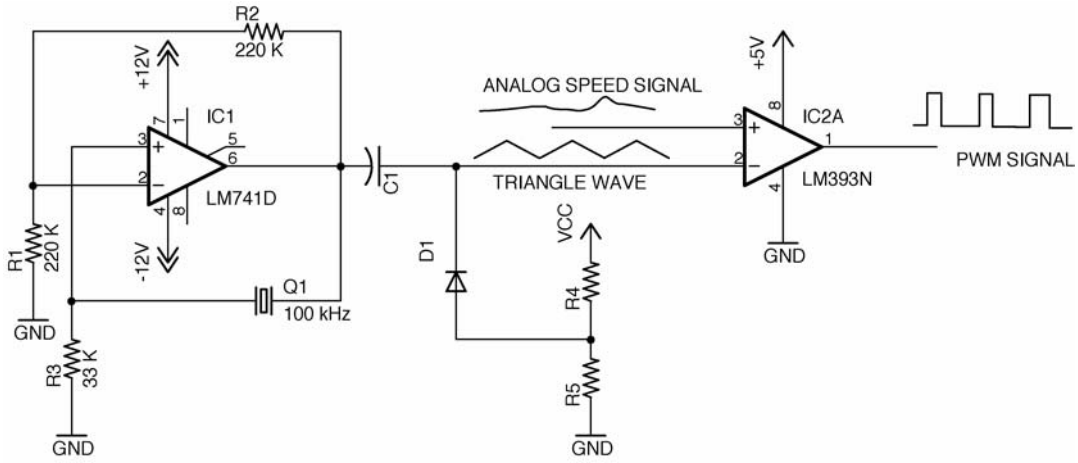

Figure 12. Crystal Oscillator PWM Circuit.

Ultimately, the crystal tank circuits' frequency proved too high to be practical. The 2531 optoisolator IC available for use on the robot distorted the signal above 10 to 11 kHz due to insufficient slew rate. Further, it was discovered the motor controller manufacturer recommended an input signal frequency of 1 kHz.

c. Final PWM Circuit using RC Oscillator

Inductor resistor capacitor (LRC) tank circuits were unsuccessfully tried and discarded. In the end resistor capacitor (RC) oscillator circuits were investigated because of their lower frequencies. A complimentary metal oxide semiconductor (CMOS) relaxation oscillator using LM741 opamps was tested. Its triangle wave output was not symmetric nor was the slope, despite trimming the opamp. Research provided a good example that formed the basis of the circuit that was installed on the Creature [21]. It used one quad LM324 opamp in place of the two LM741 opamps. To better accommodate the single voltage supply of the robot and increase the triangle wave's output voltage range, a LM6132 10 MHz, rail-to-rail opamp was substituted for the LM324.

Figure 13 shows a portion of the complete circuit that was manufactured and installed. It shows one channel and omits the portion of the circuit responsible for optical isolation. The circuit proved compatible with the single supply electrical power available the on the robot's electronics power bus owing to the use of the voltage divider reference voltage used as inverting input into opamp B. The LM6132 opamp B is connected as a Schmitt trigger and acts as an astable multivibrator. The opamp A acts as an integrator on the first opamp's output. It provides delayed negative feedback to the Schmitt trigger opamp. Because the Schmitt trigger's output is a constant equal to either the positive or negative rail voltage, integrating the constant voltage produces a signal whose voltage varies linearly with time.

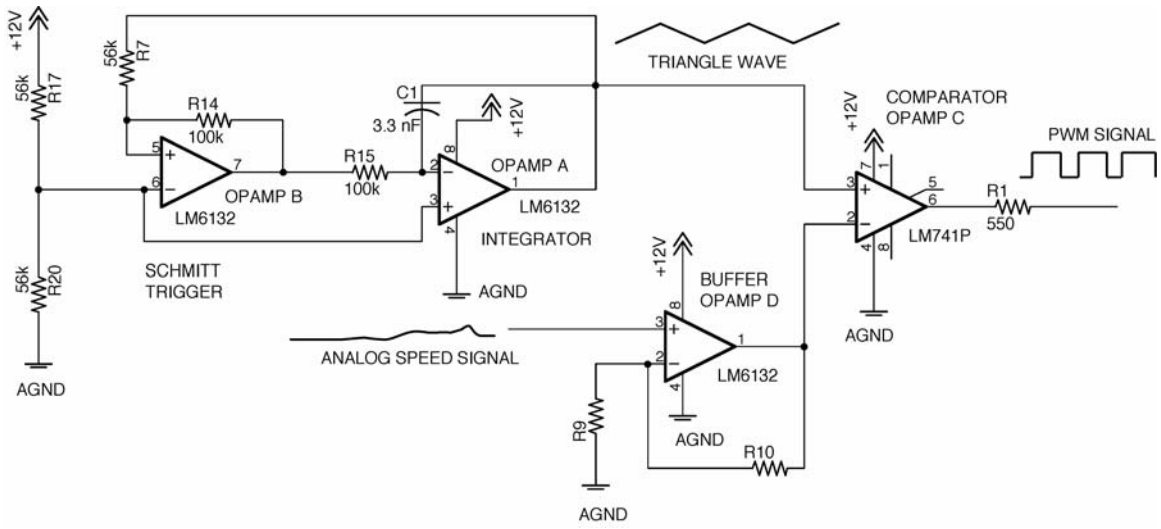

Figure 13. Single Supply Triangle Wave PWM Circuit.

The frequency of the triangle output is a function of the resistor R15, capacitor C1, and the Schmitt trigger's thresholds. The circuit produced a PWM frequency equal to 1.33 kHz. The resultant triangle wave was supplied to a LM741 opamp C configured as a comparator.

The robot's CPU, initially the BL2000, had an analog voltage output range 0 to 4.096 V, so non-inverting amplifier was included in the circuit. Opamp D was configured as a non-inverting amplifier with a gain equal to two. The gain was selected so that an analog voltage equal to 4.096 V would coincide with the maximum voltage of the triangle wave and produce a 0% duty cycle PWM signal. Resistors R9 and R10 were chosen to be 18 kΩ and 23.1 kΩ, respectively. The opamp had the added benefit of providing high input impedance for the CPU's signal.

The PWM signal produced by the portion of the circuit above was passed to dual 2531 optoisolators next. Each device contains two emitter light emitting diodes (LEDs) to convert the electrical signal into an optical one and a detector to reverse the process and produce an electrical output voltage. Figure 14 shows a portion of the circuit responsible for isolating one channel. A capacitor, C5 was installed for noise suppression. In this configuration, the 2531 was connected to produce negative pull-up.

Note, the PWM circuit above produced a 100% duty cycle when the analog speed signal was zero. The duty cycle decreased with increasing analog voltage. The negative pull-up, or pull down, applied to the optoisolator inverted this, and the resulting duty cycle of the optoisolator's output was 0% when an analog speed signal of zero was applied. It increased linearly with increasing voltage once the voltage exceeded the 1.2 V threshold. A unity gain amplifier was included between the optoisolator and the circuit's output to impedance match the motor controllers' input.

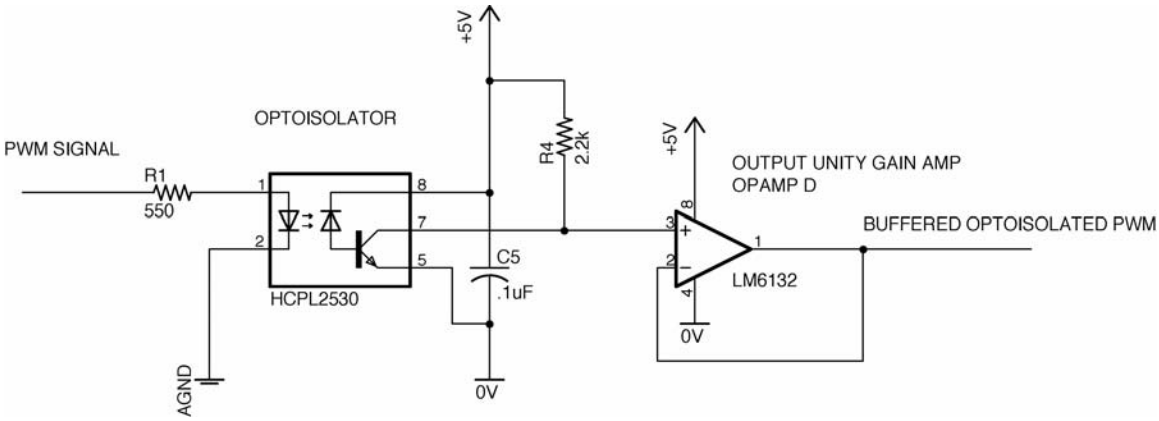

Figure 14. Optoisolation and unity gain amplification of PWM signal.

The resistors R1 and R4 were chosen to provide the correct forward current, I_f, through the optoisolator. Use of the LM741 opamp as a comparator ensured that the circuit could tolerate R1 resistance of 550 Ω without dragging down the PWM voltage due to impedance mismatching. Resistor R1 was chosen such that I_f was equal to 10 to 15 mA with PWM signal voltages 8 to 10 V peak to peak. Selecting R4 equal to 2.2 kΩ ensured sufficiently high output voltages. The optoisolated PWM voltages were observed to be 4.9 V peak to peak.

The circuit also passed three signals that could be used for either brake or direction signals to the motor controllers. Because of the negative pull up in the optoisolators, if a high logic level was desired at the motor controller, then a zero volt signal to the PWM-generating circuit was required. Of note, although the LM6132 "rail-to-rail" opamps were used to buffer the PWM speed signals, a quad LM324 opamp was

used to buffer the direction signals to reduce the number of components on the PCB and save space. This choice proved adequate, but the voltage of the high logic state was observed to reach a mere 3.7 V, not 5.0V. Regardless, the signals' voltages were sufficiently high to properly operate the motor controller.

The complete circuit is provided in Appendix A. It shows the three motor channels, optical isolators and unity gain amplifiers installed to buffer the output. The PCB for the three channel circuit was laid out using Cadsoft's Eagle circuit design software, which produced all the needed Gerber and drill files for manufacturing. The PCB was installed under the Main Power Panel (MPP) to limit the length of the motor power bus leads. The optoisolated PWM signals leave the PWM board and are routed to the underside of the chassis as a RFI reduction measure.

C. ELECTRICAL POWER SYSTEM

1. Design Goals

The electrical power system was designed to be functional, reliable, expandable, and compact. Part of functionality and reliability was the desire to provide the CPU and electronics with supply voltages that were noise free. At the earliest stage of design, it was decided that separate power supplies for motors and electronics were necessary to achieve low noise on the electronics supply. No previous SMART program robot had implemented isolated battery power supplies [9,10,11]. Reducing or mitigating RFI was a reliability concern, since noise supply voltages could interfere with reliable operation of sensitive microcontroller devices. Fabrication of electrical bus components relied heavily on handmade printed circuit boards to improve reliability by eliminating wire runs that could act as antennas for RFI. PCBs also helped achieve neat, compact components that fit the Creature's space constraints. Power connectors in excess of the planned number of loads were included in the design to provide expandability.

2. Electrical System Design Evolution

The Creature's electronics power supply underwent three major revisions. The earliest power bus included a forerunner of the current MPP. Constructed using heavy barrier strips to connect to the robot's three batteries, the first bus' 700 g mass and large size rendered it unsatisfactory when the battery arrangement was changed. The Main Power Panel, constructed from printed circuit board materials, followed.

Initial mockup of the Creature's major components called for three identical 12 V Nickel Metal Hydride (NiMH) batteries to be symmetrically mounted under the chassis. Two were planned to supply power to the three DC motors, and one was to provide electronics power. This elegant arrangement ensured the center of mass coincided with the chassis' center. The linear regulators responsible for the twelve-volt power required roughly two volts headroom above the regulated voltage, though, and the original electronics battery was not able to provide sufficient voltage. The NiMH battery was removed and an extensive redesign of the electrical system and chassis ensued.

Experience with earlier SMART program robots indicated the largest electronics load would be the robot's wireless communications device [11]. The current demand of the two routers available to the researcher was anticipated to be of the order of one Ampere, and this relatively large current demand ultimately caused problems with the first-generation electronics bus. A second-generation electronics power bus and a new buck switching DC/DC regulator were designed to address the shortcomings of the first-generation bus. The 5V DC/DC regulator section provides further details regarding the theory and construction of the buck switching regulator.

Appendix B provides an index to numbered electrical supply lines, digital signal lines, and analog signal lines that are installed in the robot. Color-coded wires also have been numbered for ease of reference. Consult Appendix B for color codes as well.

3. Overview of Motor and Electronics Power Busses

The Creature's electrical power distribution consists of an electronics power bus and a completely isolated motor power bus. This isolation was implemented to prevent noise from the DC motors producing noise on the power and ground lines of sensitive

electronics such as the CPU or microcontrollers. Figure 15 shows a simplified block diagram of the major components of the robot's electrical system. Yellow components represent electronics bus components, while green is used for motor power bus devices. The following components have separate electrical supplies and different ground voltage references:

- Main Power Panel (MPP)
- Pulse Width Modulation (PWM) and Optoisolation Circuit
- Test and Distribution Panel

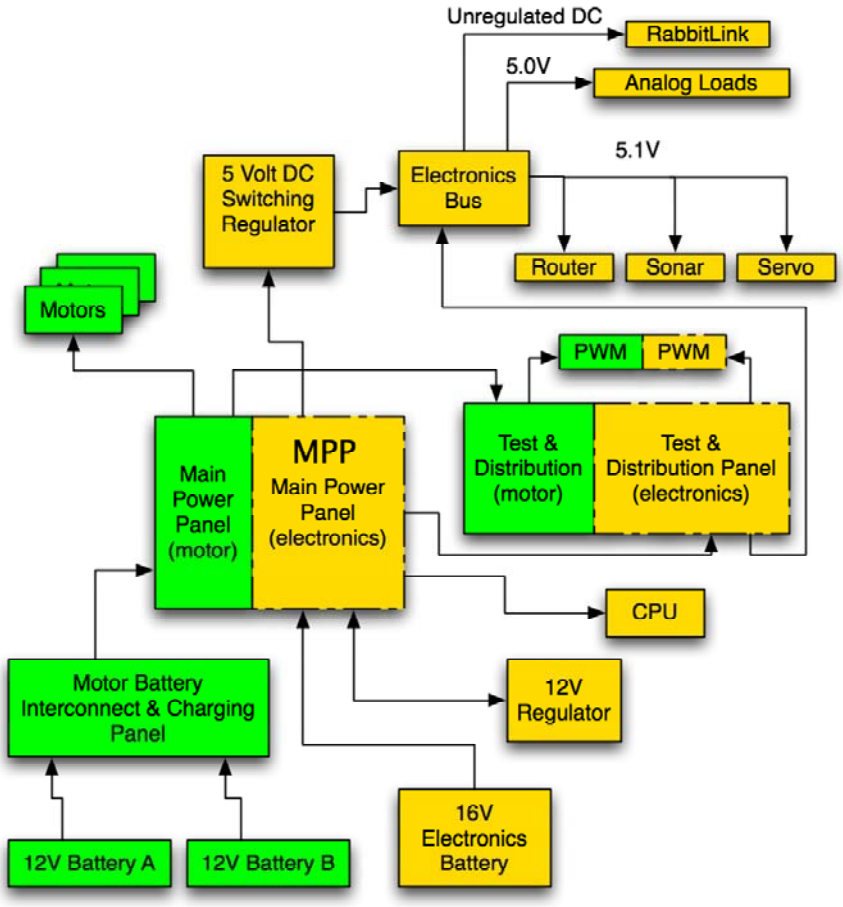

Figure 15. Simplified Electrical System Diagram.

4. Motor Power Bus

The motor power bus is expressly designed to provide unregulated voltage to three loads, namely the three DC motors for propulsion. It has a secondary function to provide 5V DC power to the optically isolated components in the PWM circuit that must be isolated from the electronics power bus. To the maximum extent possible, all wiring runs for the motor power bus are located underneath the chassis' disk. Where connections must be made to components atop the chassis, such as the Test and Distribution Panel or Main Power Panel, the length of the leads on the upper side of the chassis has been minimized by routing them under the chassis to points near the component before routing the wires back up through the chassis. The motor power bus consists of four components:

- Batteries
- Motor Battery Interconnect and Charging Panel
- Main Power Panel (MPP)
- Test and Distribution Panel (TDP)

a. Motor Power Batteries

Two NiMH batteries with Anderson Power Products (APP) Powerpole connectors are connected to a motor battery interconnect panel mounted under the chassis. The APP connectors are color-coded red positive and black negative. The connectors provide secondary electrical isolation; the interconnect panel provides the primary electrical isolation. Each twelve-volt battery consists of 10 C-cells, and has a 4000 mAh capacity. Each battery's mass is 772 g. Fully charged, each battery has been observed to possess roughly 12.8 to 13 V potential between terminals.

b. Interconnect and Charging Panel

The interconnect and charging panel's function is to allow for independent charging of the two batteries. The panel is a printed circuit board (PCB) with a 1 oz copper layer using 0.15 inch traces. It mounts two DPDT switches. Wide traces were used to allow for expected current flows of approximately 4 to 6 A. Figure 16 shows a schematic representation of the panel.

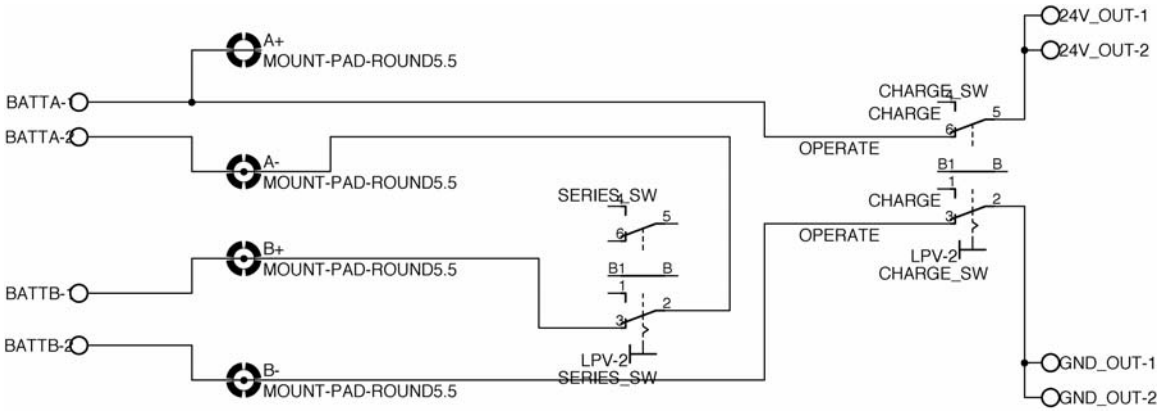

Figure 16. Interconnect and Charging Panel Schematic.

On the left of the diagram, the connections represent screw terminals which, in turn, are connected to short intermediate leads with Anderson Powerpole connectors to interface with the two batteries' leads. When the panel's series switch is closed, it allows the operator to put the batteries in series, providing 25 to 24 V, depending on battery charge. The panel's charge switch connects the batteries in series to the motor power bus. Opening the connect switch disconnects the batteries from the bus for independent charging if the series switch is opened, too. The panel includes "banana" jack connections for connecting a battery charger to each battery without the need to remove the motor batteries or disconnect battery leads.

c. Main Power Panel (MPP) Motor Power Bus Section

The MPP is a PCB mounted atop the chassis. Its primary function is to switch the main electrical supplies and provide current protection with fuses. Referring to Figure 15, one can see the board contains a motor power bus section and an electronics power bus section. The following description refers to the panel's operation with respect to the motor power bus. Function of the panel with respect to the electronics power bus is addressed separately. Figure 17 shows a schematic of the motor power bus section of the MPP. Although collocated on the same PCB, electrical isolation demands that the components not share any common leads or traces, and the traces were intentionally grouped to increase separation. To properly handle the expected currents to the motors,

the motor traces were kept wide, typically 0.1 inch. Although 1 oz copper PCB material was used in the board's construction, 2 oz material would be preferred to increase the current carrying capacity and limit Ohmic heating. Two 330 µF capacitors, labeled C1 and C2 in Figure 17, limit noise on the unregulated series voltage output.

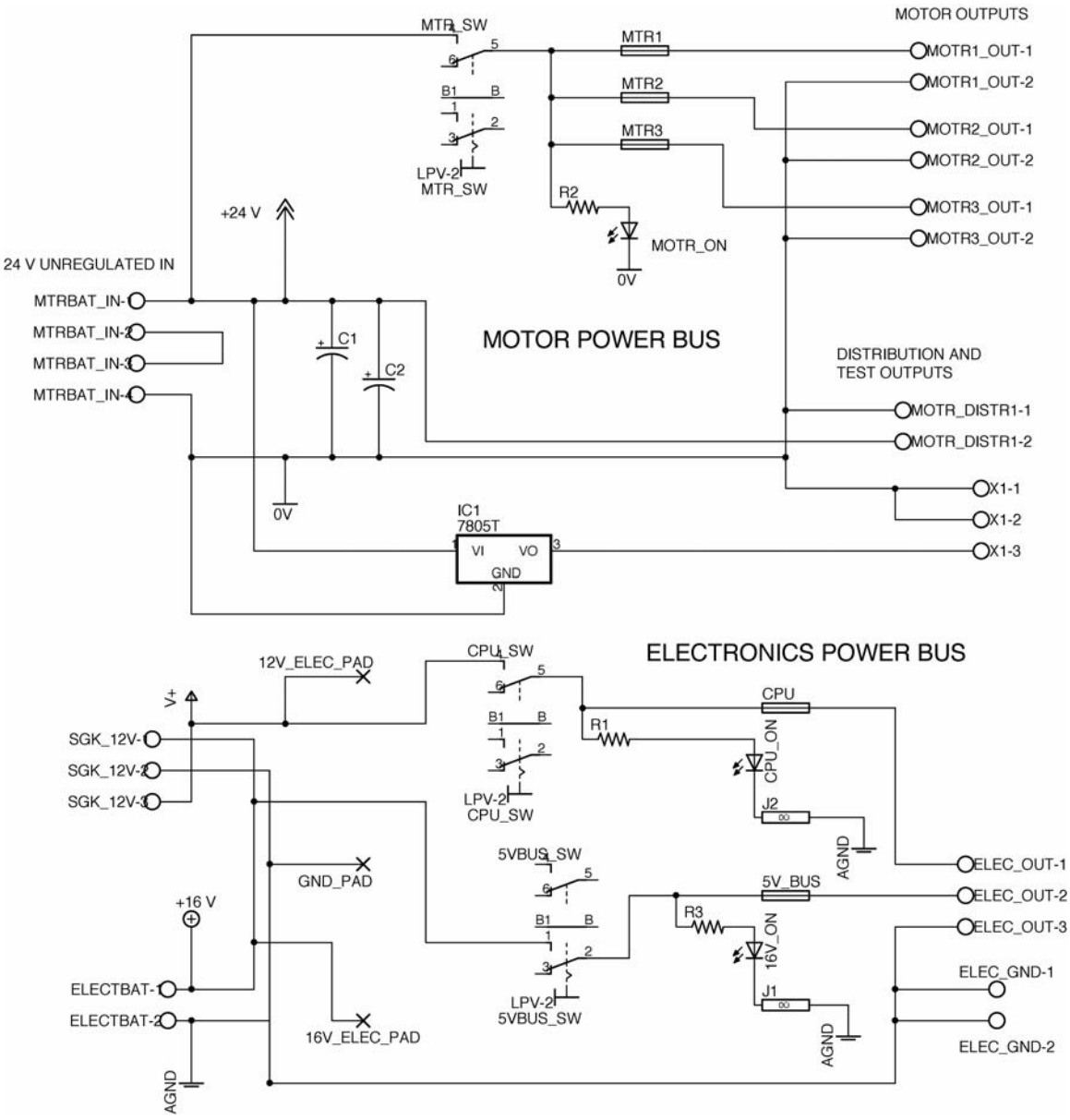

Figure 17. Main Power Panel Schematic.

The MPP includes a double pole double throw (DPDT) switch for the Motor Power Bus. Closing the switch connects the unregulated series voltage input from the interconnect and charging panel to the three motor controllers via three independent fuses, one for each motor. This is a precautionary measure and provides for future growth/change, since the installed motor controllers contain on-board fuses. Alternative motor controllers could be used in the future without the need to redesign the robot's power bus. Presently, 2 A fuses are installed, which limits current well below the 3 A operating limit of the motor controllers. An LED indicator lights when the motors are receiving DC voltage. Opening the motor switch opens the circuit, removing the series voltage and allowing the operator to safely replace fuses. DC voltages for the motor controllers are output through screw terminal connections.

A secondary function of the panel is to provide five volt DC power to components in the PWM circuit that must be electrically isolated from the electronics power bus's five volt bus. To accomplish this, the main power panel mounts one five-volt LM7805 linear regulator in a TO-220 package. It is heatsinked, although the current draw by the PWM circuit's components does not demand it. The MPP has screw terminal connections for connecting its regulated five volt supply, unregulated motor battery series voltage, and 0 V ground reference to the Power Distribution and Test Panel. Motor power is supplied directly to the individual motor controllers.

d. Test and Distribution Panel

The Test and Distribution Panel contains sections for the motor power and electronics power busses. The panel's function is to provide easy access to motor bus voltages for testing and troubleshooting. Additionally, the panel contains four banks of WAGO connectors. Two of the banks are connected to motor power bus sources and allow the user to supply electrical power to devices installed on the robot. The motor power bus' primary loads, the three DC motors, receive their supply separately, directly from the MPP. The bus has only one other load. It supplies five volts to a portion of the

PWM and optoisolation circuit containing the 2531 optoisolators and opamps that are detailed in the section describing the PWM circuit. Figure 18 shows a diagram of the test and distribution panel.

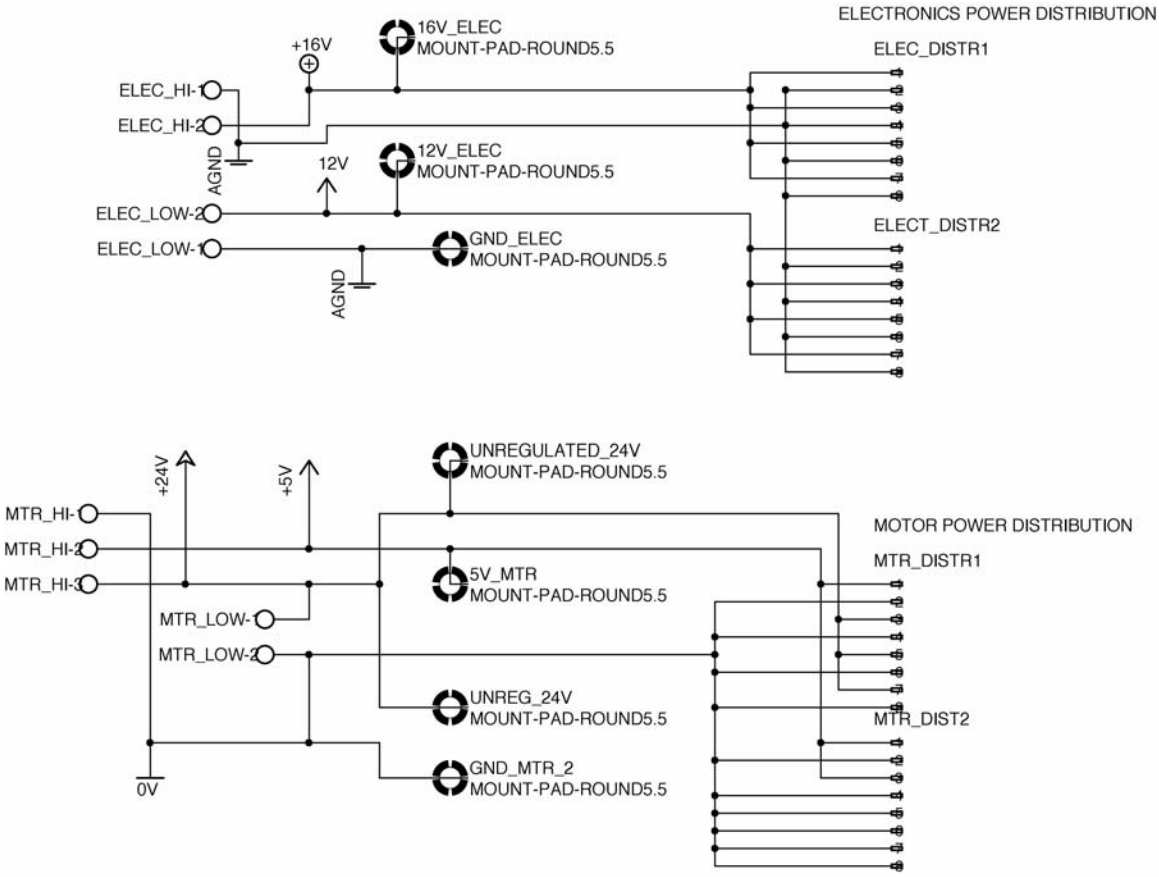

Figure 18. Test and Distribution Panel Schematic.

5. Electronics Power Bus

The electronics power bus consists of seven components:

- PowerStation 100 Electronics Battery
- AC Power Supply
- Main Power Panel
- 12V Regulator Panel
- Test and Distribution Panel

- 5V Electronics Bus Panel (First-generation)
- 5V Electronics Bus Panel (Second-generation)
- 5V DC Buck Switching Regulator Panel

Consulting the simplified system diagram presented in Figure 15, one should note that the electronics power bus loads fall into four major categories. Loads with sensitive analog components such as the Inertial Measurement Unit (IMU) operated best with precise 5.0 V supply. Other devices such as the sonar sensor head controller, namely a PIC microcontroller, can operate on a range of voltages around 5.0 V. A third class of loads, such as the wireless router, require five-volt regulated supply, but can function with small variations, e.g. 0.1 to 0.2 V, about 5.0 V. The fourth class of loads includes loads that have on-board voltage regulation. The BL2600 SBC and RabbitLink programmer are capable of self-regulation of the supply voltage.

The table below lists the installed electronics devices, their supply voltages, the observed current demand measured at the output from the electronics bus during normal robot operation, and the power required based on the equation below. The power required by each device, P, is equal to the measured current, I, multiplied by the voltage, V.

$$P = I \cdot V$$

Device	Current (mA)	Voltage (V)	Power (W)
Router	700.0	5.1	3.6
BL2600	210.0	12.0	2.5
RabbitLink	75.0	16.5	1.2
Sonars & PIC controller	215.0	5.1	1.1
Sonar Sensor Head Servo	180.0	5.1	0.9
IR Rangers (6)	159.0	5.0	0.8
PWM	61.0	12.0	0.7
IMU	96.0	5.0	0.5
Tachometer & Detectors	61.0	5.0	0.3
I2C Compass	8.8	5.0	0.0
I2C SCL & SDA	0.4	5.0	0.0

Table 1. Measured Electronics Bus Loads.

a. Electronics Battery

The original electronics battery was replaced with a PowerStation 100 lithium ion battery manufactured by Total-micro. It was designed to provide DC voltage to laptop computers consuming 60 to 75 W. According to the manufacturer, the battery's specification calls for it to provide 16 to 20 V under 6.6A load, which agrees with its nominal 16.5 V rated voltage [22, 23]. According to its specification, its rated capacity is 6.6 Ah [23]. Its mass is 907 g, and it measures 0.7 cm x 21.8 cm x 29.0 cm. The battery is connected to the electronics power bus section of the MPP using the manufacturer's #6 adapter plug, which provides center positive and exterior ground.

b. AC Power Supply

A Sony AC-V018G AC to DC adapter was found to supplement the PowerStation 100 battery. If prolonged stationary testing of the robot's sensors or CPU is required, the operator can disconnect the electronics battery and connect the AC power supply in its place to preserve the battery's charge. The device's case states that it is rated at 18V and 4A, 72W maximum. Its original connector was replaced with a plug matching the PowerStation's #6 plug, with a center positive and exterior ground.

c. Main Power Panel (MPP) electronics bus section

The Main Power Panel contains an electronics bus section and a motor power bus section. This paragraph describes its functionality with respect to the electronics bus. The panel contains switched fuses to provide current limiting for electronics. The panel contains two DPDT switches. Figure 17 shows a schematic of the MPP. The electronics battery's leads are connected via screw terminal connectors to the panel. The unregulated battery voltage is constantly connected to the 12 V Regulator Panel. When the CPU switch on the MPP is closed, the 12 V Regulator Panel's output passes through a fuse and is connected to a screw terminal ELEC_OUT-1. From the terminal the regulated twelve volts is routed to the Test and Distribution Panel. An LED lights when the switch is closed to indicate the CPU is receiving power. A second

switch, labeled 5VBus is located between the CPU and motor switches. When it is closed, unregulated power from the electronics battery is allowed through the fuse and routed to screw terminals for connection to both the 5V/12V Bus Panel and the Test and Distribution Panel. An LED operates to indicate power is available to the bus panel.

d. Test and Distribution Panel

The Test and Distribution Panel contains sections for the motor power and electronics power busses. This section describes its operation with respect to the electronics power bus. The panel provides easy access to electronics bus voltages for test and troubleshooting. Additionally, the panel contains two banks of WAGO connectors on its electronic power bus side. One provides unregulated voltage (nominally 16 V) from the electronics battery, and the other provides twelve-volt power. The CPU is the only twelve-volt load connected to the panel. It is switched via the MPP's CPU switch. Unregulated voltage is available to the panel's WAGO connectors when the MPP Bus switch is closed. The RabbitLink Ethernet programming interface for the BL2600 CPU is the sole load for unregulated voltage from the Test and Distribution Panel. It receives its supply via the third generation 5V Electronics Bus.

e. 12 V Regulator Panel

Although most electronic devices on the robot required five volts, the robot's CPU and original router required twelve-volt power. The space available on the chassis and the need to reserve the area over the center of the robot for the IMU required a separate panel for producing regulated 12 V. A compact PCB was built to mount one SGK LM7812 linear regulator in TO-3 package, a heatsink, and screw terminal connectors. The PCB included a generous ground plane on the reverse to reduce noise. It was positioned to minimize the length of the leads connecting it to MPP. The MPP supplies a nominal 16 V from the electronics battery to the regulator, and the regulator's output is returned to the MPP where it is switched. Figure 19 shows a photograph of the panel. The edge of the MPP is on the left, and the 12 V Regulator Panel is in the center.

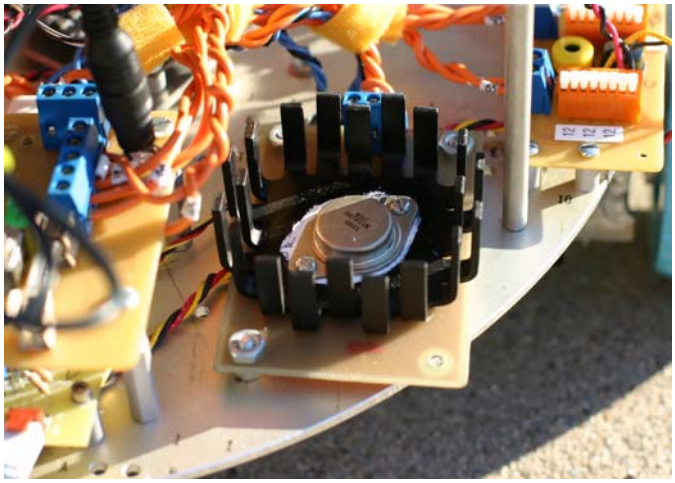

Figure 19. Twelve Volt Regulator.

f. 5V Electronics Bus Panel (First-generation)

Initially, the Creature's electronics were planned to use exclusively five-volt devices. The first-generation bus panel provided supply voltage and ground connections for electronic devices installed on the Creature. In keeping with the electrical system design goals of expandability and reliability, a PCB was constructed to mount four LM7805 linear voltage regulators and seven banks of WAGO connectors. LED indicator lights were omitted to reduce voltage drops. The number of WAGO connectors was chosen to provide numerous unused connections for future loads. The output of each of the regulators is switched. The sonar modules can draw momentary peak currents of approximately 1 A during transmit [24]. The SHARP infrared range sensors were selected to share the regulator with the sonar modules because of the IR sensors' small current consumption. Devices using the I2C bus were teamed to receive five-volt power from a single regulator. Due to the large current demand of the wireless router it was provided its own linear regulator.

The completed first-generation bus panel proved inadequate and required a redesign. The LM7805 regulator failed to maintain voltage when loaded by the wireless router. The board's layout impeded easy access to three of the banks of WAGO connectors. Its usability was mediocre: the placement of the WAGO connectors did not allow users sufficient room to easily connect wires.

g. 5 V DC Buck Switching Regulator Panel

The failure of the first-generation electronics bus to maintain its five-volt output when under load prompted the researcher to address the root of the problem and develop a robust five-volt supply. In keeping with other work on the Creature, efforts were made to ensure the solution was reliable and extensible to current and future robotics projects in the SMART program. Noting that the original LM7805 had been producing excessive heat, in effect wasting battery power to produce heat as a byproduct, the researcher decided to pursue a more efficient DC/DC conversion method. Switched power supplies are commonplace in all manner of battery-operated devices, from laptop computers to cellular phones, but none had been installed previously on a SMART program robot. Equation 3.1 from Pressman gives the linear regulator's efficiency [25]. Equation 3.2 gives the efficiency of a buck switching converter [25].

$$Series\ Pass\ Device\ Efficiency = \frac{P_o}{P_{in}} = \frac{V_o I_o}{V_{dc} I_o} = \frac{V_o}{V_{dc}} \tag{3.1}$$

$$Buck\ Switching\ Converter\ Efficiency = \frac{P_o}{P_o + DC\ losses + AC\ losses} \tag{3.2}$$

Implementing a simple step down, or buck switching, DC/DC converter could result in drastically improved efficiency as demonstrated below, where V is the voltage, P the power, and I the current. If one assumes a 1 V diode drop across the diode and a worst-case AC switching loss case, then the losses are [25]:

$$DC\ losses = V_{diode} I_o = 1 I_o$$

$$AC\ losses = 2 V_{dc} I_o \frac{T_s}{T}$$

And the DC/DC converter efficiency becomes [25]: :

$$Buck\ Switching\ Converter\ Efficiency = \frac{V_o I_o}{V_o I_o + V_{diode} I_o + 2 V_{dc} I_o \frac{T_s}{T}} = \frac{V_o}{V_{out} + 1 + 2 V_{dc} \frac{T_s}{T}}$$

Consider an example where the linear regulator's only load is a wireless router, drawing current equal to 0.7 A, as in Table 1. If the linear regulator's negligible quiescent current is neglected, then I_{out} equals I_{in}. The efficiency reduces to a ratio of the output to the input voltage. In the case of the first-generation electronics bus, stepping down an input voltage of 16V to 5.0 V gives an efficiency of 31%. In contrast, a buck switching regulator operated at 22 kHz with a switching delay, T_s, equal to 0.3 μs would step down the same input voltage with efficiency equal to 82%. Note, Pressman's treatment assumes the voltage drop across the diode is 1 V. Using a Schottky diode, the voltage drop V_{diode} would decrease from 1.0 V to roughly 0.6 V, and efficiency would increase.

Whereas a linear regulator operates a series pass transistor in its linear region, effectively using the device as a variable resistor, a buck switching regulator operates on an entirely different principle. As the name implies, a switching regulator rapidly switches a fast-operating transistor on-and-off. For a constant period, increasing the switching duration of the on portion of the period produces a pulse width modified duty cycle. The time average of the voltage output increases with the increased on period time. To smooth the switch's square-wave output, an inductor-capacitor (LC) filter is used. The LC filter outputs are "ripple-free DC voltages equal to the average of the duty-cycle-modulated raw DC input…." [25]. Alternatively, one can think of the inductor as storing energy in its magnetic field while the switch, or transistor, is closed. When the switch is opened, the magnetic field's energy is released into the load [26].

Figure 20 shows a circuit diagram of an example buck switching regulator. The MOSFET Q1 serves as the switch. The inductor L_0 acts in concert with the capacitor C_0 to filter the switched voltage.

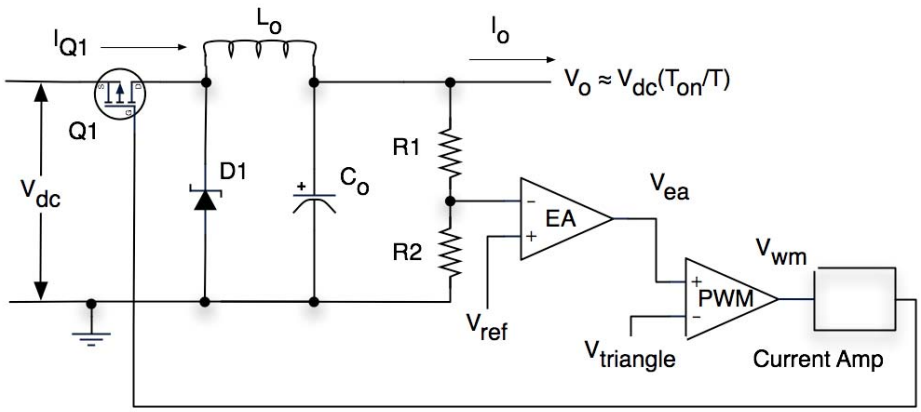

Figure 20. Example Buck Switching Converter. (After Pressman)

Assume an ideal case where the voltage drop across the MOSFET is zero. When the MOSFET acting as the switch is closed, the voltage potential across the inductor L_0 equals V_{dc}- V_o. The change in current with respect to time is given by Equation 3.3.

$$\frac{dI}{dt} = \frac{(V_{dc} - V_o)}{L} \tag{3.3}$$

Since the difference between input and output voltages is a constant as well as the inductance, the current trough the MOSFET behaves in a linear manner producing the positively sloped current I_{Q1} waveform (d) in Figure 21. One can see the current waveform (d) increase linearly until time T_{on}. After time T_{on} the switch is opened, turning off the voltage supply.

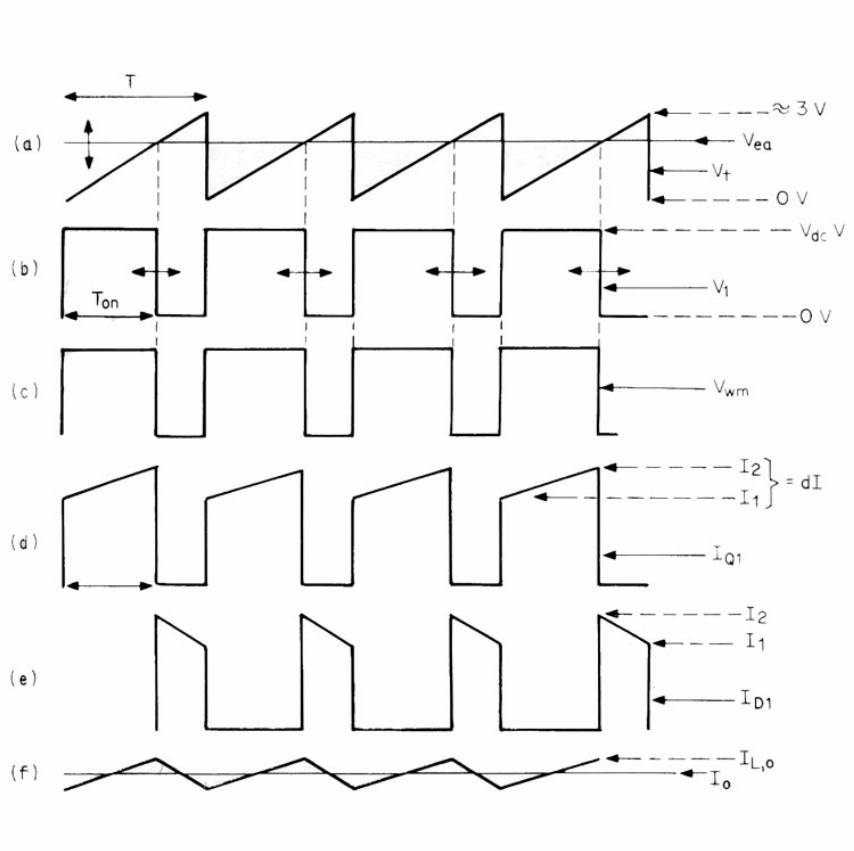

Figure 21. Buck Switching Regulator Waveforms. (From Pressman)

While the MOSFET switch is closed, the increasing current predicted by Equation 3.3 creates an increasingly large magnetic field. When the MOSFET is opened at T_{on}, the magnetic field present in inductor L_0 is still present; it cannot instantly disappear. Rather, it resists the change in current, but the open switch prevents any current from flowing from the battery through the MOSFET. Instead, the diode D1 initially supplies an equal current I_{D1} shown as waveform (e) in Figure 21.

The diode only allows forward current flow and the anode is at a zero potential with respect to ground. Thus, the cathode must be more negative than zero. In other words, the cathode must be at a negative potential equal to one diode voltage drop, V_{diode}, below ground. The output voltage, V_o, is unchanged, so the potential across the inductor is the difference between the output voltage and the negative diode drop voltage. The difference is equal to $V_o - (-V_{diode})$. Because the polarity has been reversed, a negative sign is introduced, and current through the inductor is now given by the Equation 3.4.

$$\frac{dI}{dt} = \frac{-(V_o + V_{diode})}{L} \tag{3.4}$$

Now the current flowing through the diode linearly decreases with time at a rate proportional to the sum of the output and diode drop voltages and inversely proportional to the inductance. Let I_2 be the highest current achieved with the switch closed and I_{Q1} be defined as the lowest current when the switch is open. Combining the waveforms (d) and (e) in Figure 21 produces a current waveform (f) that varies by $I_2 - I_1$ about a center value I_o.

A combined MOSFET and regulator IC, the L4964, by STMicroelectronics was selected as the basis of a DC/DC regulation circuit. It is a high-current device capable of supplying 4 A and able to produce a wide DC output voltage ranging from a preset 5.1 V to 28 V. It requires few external components, and its switching frequency is easily adjustable using an off-chip RC circuit up to a maximum of 125 kHz. A toroidal 138 μH inductor by Coilcraft, model DMT-3-138-6, was selected as the regulator's inductor. The toroidal design was favored because of the toroid's inherent magnetic field shielding. Additionally, the inductor was selected to because its saturation current, 6.0 A, was greater than the expected maximum current produced by the regulator.

The switching frequency, f_{sw}, was chosen using the equation below which assists engineers in choosing inductors for buck switching supplies [27]. Based on the Creature's electronics battery voltage, the maximum input voltage, $V_{in\ max}$, was set to 16.5V, and the output voltage, V_{out}, was set to 5.1V. LIR, the inductance to current ratio,

was recommended to be 0.3, but lower values would produce a more efficient regulator at the expense of transient current response. For the Creature, an LIR equal to 0.3 was used, and maximum current, I_{max} , was set to 4 A. The appropriate switching frequency was determined to be 21 to 22 kHz.

$$f_{sw} = \frac{(V_{in\max} - V_{out}) \cdot V_{out}}{V_{in\max}} \cdot \frac{1}{L} \cdot \frac{1}{LIR \cdot I_{\max}}$$

The output capacitance depends on the output ripple voltage allowable. The peak current in the inductor is I_{peak} and given by the equation below [27]:

$$I_{peak} = I_{\max} + \frac{1}{2}\Delta I_{inductor} = I_{\max} + \frac{(V_{in\max} - V_{out}) \cdot V_{out}}{V_{in\max}} \cdot \frac{1}{L} \cdot \frac{1}{f_{sw}}$$

Based on the designed 4A maximum current and 22 kHz switching frequency, a peak current of 4.4 A was expected. Thus, the Coilcraft inductor above was deemed acceptable.

The output capacitor, C_o, in Pressman's diagram, was chosen using the equation relating output capacitance to voltage overshoot, ΔV [27]. Given the values above and a desired voltage overshoot of 0.1 V, the output capacitance was determined to be 2600 μF.

$$C_o = \frac{L \cdot (I_{peak})^2}{(\Delta V + V_{out})^2 - V_{out}^{\ 2}}$$

The buck switching controller, toroidal inductor, output capacitors, Schottky diode, and ancillary circuit components are shown in Figure 22, which provides a circuit schematic of the DC regulator. A Motorola MBR1635 diode, a device designed expressly for applications such as this one, was chosen for its ability to pass the expected current. Four 330 μF capacitors in parallel were used in place of a large single capacitor

to reduce the equivalent series resistance. Although the design called for 2600 μF output capacitance, a larger voltage ripple was deemed acceptable in order to avoid the large series resistance that such a capacitor would have created had it been installed.

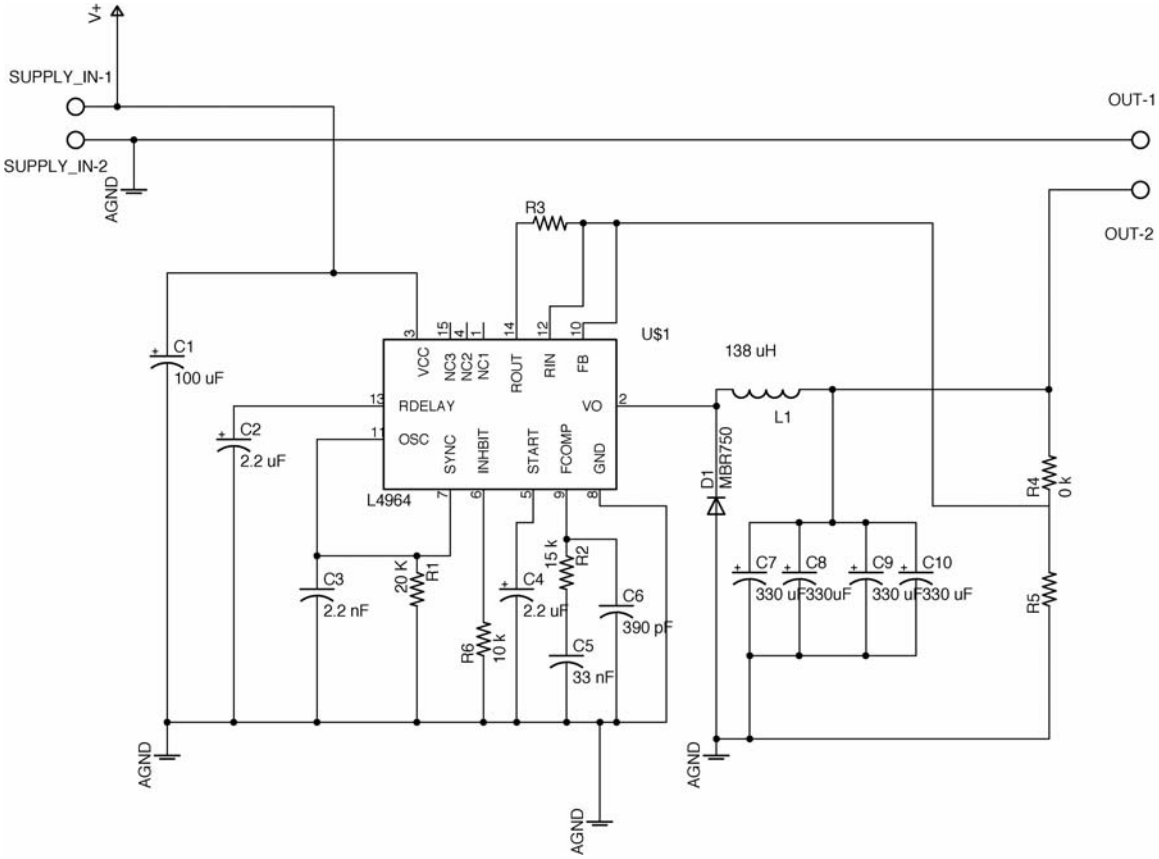

Figure 22. DC/DC Buck Switching Regulator Schematic.

h. 5V Electronics Bus Panel (Second-generation)

Coincident with the construction of the DC/DC switching regulator, a replacement 5V electronics bus was designed and built by hand from PCB. Whereas the first-generation bus combined the voltage regulation and load switching functions on one PCB, the DC/DC buck switching regulator circuit required its own circuit board. Thus, the functions were separated between two PCBs in the second-generation bus. A side-effect of this decision was that after the DC/DC regulator was proven, duplicate PCBs could be made and used on any SMART program robot. In fact, this came to pass.

When the servo motors operating the Bigfoot robot's mechanical arm required a high-current 6 V supply, a duplicate of the Creature's regulator was pressed into service [28].

Although the Creature's high-current five-volt loads were capable of operating off the DC/DC switching regulator's output, it was decided to provide the more sensitive sensors, specifically those with sensitive analog devices such as the IMU, a linear voltage regulator supply. The second-generation bus panel includes screw terminal connections for the regulated voltage from the switching regulator and for unregulated electronics battery voltage. Five banks of WAGO connectors, each with eight connection points, are provided for connecting electronics loads. The first-generation bus' bulky DPDT switches were replaced with single pole double throw (SPDT) ones, allowing the board to mount seven switches in a more compact layout. Deficiencies noted in the earlier board were fixed, e.g. seating wire leads into the WAGO connectors can be easily accomplished due to the WAGO's orientation. A schematic of the second-generation electronics bus panel is provided in Figure 23.

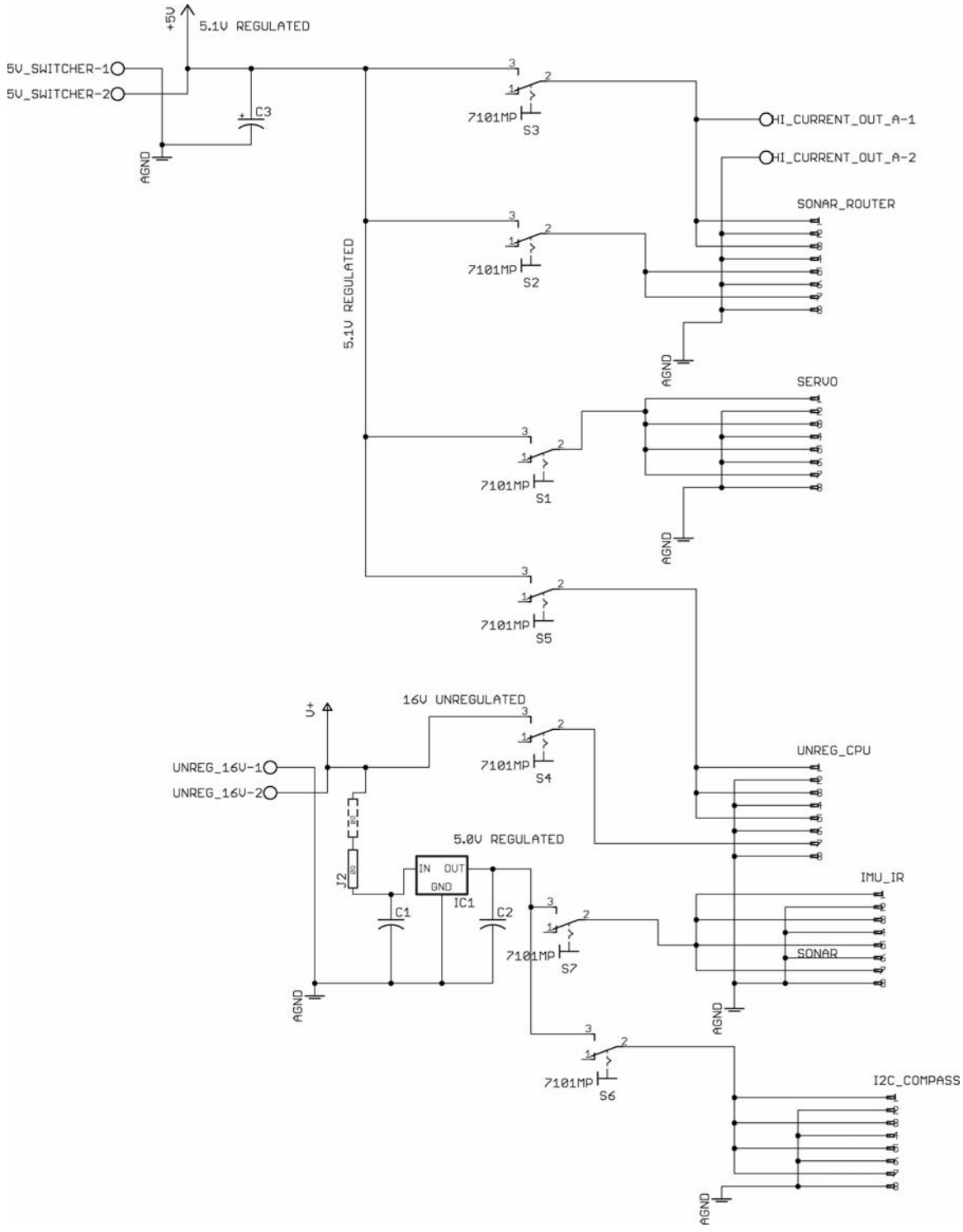

Figure 23. Second-generation Electronics Bus.

D. MICROCONTROLLERS

1. Z-World BL2600 Single Board Computer

The Creature's operating program runs on a Z-World BL2600 44.2 MHz single board computer (SBC). The computer is based on the Rabbit 3000 microprocessor, a descendant of the Zilog Z-80. It has been selected for use in other SMART program robots because of its ease of programming with the vendor's feature-rich, proprietary Dynamic C language and IDE [11]. The BL2600 includes three serial ports capable of being configured for RS232 communications, and a 10/100 BaseT Ethernet connection [29]. The microprocessor has 512 kB static RAM and 512 kB flash memory. Its 36 digital I/O are divided into sixteen digital inputs, sixteen software-configurable as input or output, and four high current outputs. The BL2600 offers several jumper-selectable options for configuring its I/O. The installed BL2600's J1 has been set to use an external 5V pull-up voltage from the electronics power bus. In addition to digital I/O, the SBC has twelve analog I/O pins, eight of which are eleven-bit A/D inputs. Most importantly, the BL2600 has four twelve-bit D/A outputs.

The BL2600 is connected to the sonar sensor head controller PCB via serial port C, which uses a three-wire RS232 arrangement consisting of a connection to ground, TxC, and RxC. The CPU is also connected to the IMU via a serial port F. It uses a three-wire RS232 and is connected to ground, TxF, and RxF. The table below provides a list of the pinouts used by the installed BL2600, their function, and the wire label number. I2C serial communications are bidirectional, and the BL2600's configurable digital I/O channels, DIO00 and DIO01, are configured as outputs. Although configured as output pins, the value of the pin can be read using library functions in Dynamic C. Because they are open collector, the outputs float to the 5 V pull-up voltage when not held low. An external 5 V pull-up voltage is supplied to the BL2600 via the +K pin. The BL2600's jumper, JP1, is set to provide open collector output using the five-volt power. This five-volt power and ground are supplied via Second-generation Electronics Bus's I2C switch. The digital output grounds are connected to ground via leads to the same circuit. The connection of the SHARP IR rangers includes two ground connections to ground on the IR switched portion of the Second-generation Electronics Bus.

BL2600 Pinout	Function	Wire Label Number
DIO00	I2C Bus SCL	41
DIO01	I2C Bus SDA	40
DIO02	Motor 1 direction	46
DIO03	Motor 2 direction	47
DIO04	Motor 3 direction	48
AV0	Motor 1 analog speed	21
AV1	Motor 2 analog speed	22
AV3	Motor 3 analog speed	20
AIN0	IR 1 analog input	1
AIN1	IR 2 analog input	2
AIN2	IR 3 analog input	3
AIN3	IR 4 analog input	4
AIN4	IR 5 analog input	5
AIN5	IR 6 analog input	6
+K	5V pull up voltage	D

2. Z-World BL2000 Single Board Computer

The BL2600 was not available initially in the Creature's development, and the robot was designed from the start to use the BL2000, which has two analog voltage outputs. Herkamp provides an explanation of its capabilities [11]. It was decided early in development to use an analog voltage signal as the speed signal for each of the three motors. It was believed that doing so would reduce CPU workload that would otherwise need to be devoted to producing three PWM signals. Pressing the BL2000 into service as the Creature's CPU resulted in an interesting problem that had not occurred previously in SMART program robots that used two analog motor speed signals for skid steering. Before motion testing could commence, a method of producing *three* analog signals needed to be conceived. To this end, a second BL2000 operating as a slave CPU was connected via RS435 for the express purpose of producing an analog signal directed by the master BL2000. This arrangement proved adequate, but a better solution was pursued during development. Installing the BL2600 reduced power consumption vis-à-vis the two CPUs and removed the failure mode of a communications loss between the CPUs, which would have resulted in restricted or possibly uncontrollable motion.

3. Microchip PIC16F690

Two devices were constructed and installed on the Creature using the PIC16F690 microcontroller manufactured by Microchip Inc. PICs are installed in the sonar sensor head controller and the wheel tachometer. This section describes the microcontrollers. For a description of the sensor devices, see the following section. The PIC16F690 was selected for its ease of programming, product support, on-board serial port, numerous pieces of example code, and widely available documentation. Although the PICs were programmed in assembly language, C language compilers were available.

Before developing a PIC microcontroller-based solution, a sonar controller PCB was designed using IC logic gates, binary counters, etc. Its component count exceeded twenty ICs, though, and its communications scheme was non-standard. A better solution using a microcontroller to reduce component count was sought. An example of the savings in component count is the wheel tachometer. In the three-wheel tachometer circuit a single PIC was used to replace an older two-wheel circuit that required over eight ICs. Also, a microcontroller solution provided flexibility in both devices' implementations. If changes to a device were needed, the device could be reprogrammed to optimize it for another mode of operation. For example, operating parameters of the sonar, such as maximum range and time delay between sonar ranging attempts could be adjusted for outdoor use by easily by modifying the code rather than building a new hardware solution. In addition, the PIC's built-in counters provided the researcher with a level of timing accuracy unmatched by the BL2600, which can only time events to 1 ms accuracy.

The manufacturer, Microchip Technology, provides a comprehensive manual on the device's operation, I/O, capabilities, and assembly language syntax [30]. Of note, the devices have three built-in counters, one with sixteen bit range. Operated at 8 MHz and 1:1 clock prescaling, the internal counters offer 0.5 μs timing accuracy. The PIC16F690 has a PWM module, a capture and compare module, and a built-in serial UART. The serial port can send RS232 signals via a level shifting IC or I2C communications.

Both PIC16F690 microcontrollers installed on the robot are operated at 8 MHz using the microcontroller's built-in oscillator. The sonar sensor head controller circuit uses one PIC16F690 communicating via RS232 at 57.6 kBd to pass its range data sentence. The wheel tachometer microcontroller uses I2C bus communications to pass its wheel speed data to the CPU when requested. The PICs' programs, in Microchip PIC assembly language, were written and compiled in the MPLAB IDE running on a Dell laptop. A PIC Kit II programmer was connected between the programming laptop and the PIC to load the programs using Microchip's In Circuit Serial Programming (ICSP), which allows the operator to program the microcontroller without need to remove it from the circuit board.

E. COMMUNICATION

Communication with the Creature is accomplished using 802.11g wireless LAN signals. Two wireless routers have been used on the Creature. In early motion testing, the CPU's robot operating program depended on Telnet commands for manual control while the researcher verified the motion portion of the Dynamic C code and the ability to produce motion as predicted in Chapter II. Initially, a Netgear Rangemax 240 router was installed under the tray holding the electronics battery. It's operation was problematic and eventually deteriorated to the point that the Telnet session could not be maintained for periods longer than roughly 30 s when transmissions were sent. Telnet sessions exceeding eleven minutes were observed when no transmissions were sent, but preserving the Telnet session by not communicating was an unacceptable condition. Investigation revealed that the Netgear router demanded excessively high current from its twelve-volt linear regulator during busy transmission periods. When the linear voltage regulator could not match the current demand output voltage to the router dropped and the device reset.

The Netgear RangeMax router was replaced with a D-Link DI-624 AirPlus Extreme router, operating on 5.0 V. The D-Link router's AC power supply is rated at 2.5 A. Such routers have been successfully used on other SMART program robots with linear voltage regulators [10]. Smaller than the Netgear router, it was easily installed in

the same location under the electronics battery tray. The router retains its original antenna, which has proven completely sufficient in testing to date. Its mass was found to be 0.3 kg.

The D-Link router was observed to draw 600 to 740 mA when operating and communicating via 802.11 g. To accommodate this it was the sole load for one LM7805 five-volt regulator on the robot's first-generation electronics bus. The D-Link router operated successfully for roughly eight weeks. Unfortunately, the heatsink attached to the LM7805 proved inadequate, and the router's power demand caused the linear regulator to heat excessively. It was believed that the heating triggered the regulator's internal thermal protection. During failures, the linear regulator's output was observed to drop below 4.90 V, which forced the router to reset itself. A long-term solution to the router's current demand is described in the switching regulator section.

F. ENVIRONMENTAL SENSORS

1. SHARP IR Range Sensors

The Creature was designed for semiautonomous operation, which required the ability to sense obstacles in the environment surrounding it. Considering the cluttered intended operating environment, the researcher chose environmental sensors based on two criteria: close minimum range, and rapid update rate. The Creature needed to be capable of exploiting its holonomic mobility with sensors that could detect obstacles as close as the periphery of the robot. The Creature was expected to operate at up to 1.0 m/s maximum. The combination of speed and cluttered environment dictated that sensors be fitted that could provide a complete scan at roughly 1 Hz to allow the CPU sufficient time to detect, slow, and avoid an obstacle. The 2Y0A02 SHARP IR ranging sensor was seriously considered because of its detection range, approximately 18 to 100 cm, and its quick refresh rate, which was observed on an oscilloscope be 40 ms. The analog output displayed discreet steps when a target at 15 cm was quickly removed from the FOV.

Herkamp provides an explanation of the operation of the SHARP IR ranger [11]. They are simple to use and produce an analog voltage inversely proportional to the range of the target. The sensor has an extremely narrow FOV that is 4 cm wide at 100 cm

range [31]. At r equal to 100 cm with s equal to 4 cm, the angular FOV, θ, would be 0.040 radians. The number of samples, N, needed to sense the obstacles at 100 cm, using θ equal to 40 milliradians, is given by the Equation 3.5:

$$N = \frac{2\pi}{\theta} \tag{3.5}$$

It would have required 120 samples to completely sense the 360º surrounding the robot without gaps. The number of samples posed a problem, but solutions using eight to ten devices atop a scanning a detector head were feasible. More troubling, the devices showed significant variation in their analog output voltage when tested in the lab against objects at a constant distance. Specifically, the output voltage was observed to be 0.7 V when viewing a Ø 0.5 inch smooth steel rod at 80 cm. The output voltage dropped to 0.37 V when the smooth rod was replaced by a Ø 0.5 inch threaded rod. A wooden meter stick placed with its widest side normal to the sensor's FOV produced a higher output voltage, 1.0 V, despite being placed at 1 m. Based on these observations the IR rangers were deemed unacceptable for use as the primary environmental sensor, and the decision was made to pursue ultrasonic sonar for this role.

Though not employed as the Creature's primary sensor, IR rangers were installed to provide a limited coverage of the space below the sonar sensor head's view. Figure 24 shows the equilateral arrangement of the IR rangers. Two IR rangers are installed antiparallel to each other and perpendicular to each of the three motor mounts. The sensors are placed at a height approximately 6 cm above the surface the robot is operating on. A total of six sensors are installed, numbered one through six per Figure 24. Sensors one and two act as a sensor pair, as do three and four, and five and six. The FOVs of the sensors cross at three locations, approximately 10 cm from the edge of the chassis at an position midway between each wheel. This arrangement provides a virtual bumper when the robot is driven in one of its three principle directions: 60º, 180º, 300º measured clockwise from the Y axis.

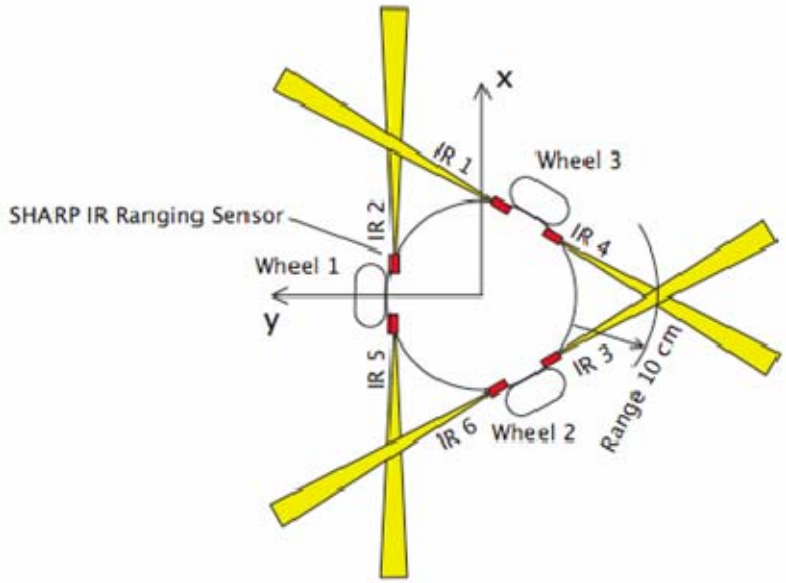

Figure 24. IR Ranging Sensor Installation.

Experimental observation of aluminum soda cans in the lab provided threshold voltage levels that the robot operating program uses to indicate the presence of an object. These cylindrical targets were insensitive to target orientation, unlike the meter stick described above. At the intersection of the sensors, output voltage was observed to be 2.2 to 2.4 V. A voltage of 1.4 V indicated the presence of an object at 50 cm from the sensor, or 33 cm from the front edge of the chassis. If the robot were moving along one of its principle motion directions, such an object might collide with the wheel on the side opposite the sensor. Threshold values were set based on observation of the robot's performance while conducting its random walk algorithm.

The six sensors' analog outputs are connected to the CPU breakout panel. The panel's function is to provide I2C bus connections and allow for easy CPU change out. For example, if future researchers install a different CPU, the IR rangers' wiring can remain intact, and the new CPU will simply need to be connected to the breakout panel. The panel also provides the sensors with five volt DC voltage for operation. The IR sensors are switched via the 5V Electronics Bus Panel.

2. Scanning Sonar Sensor Head

a. Design Considerations

The primary environmental sensor for the Creature is its scanning sonar sensor head. Although SHARP IR rangers had desirable characteristics, such as fast refresh rate and close minimum range, the inability to accurately report range and the large number of scans required to completely scan all azimuths prompted the adoption of a conventional sensor, ultrasonic sonar. The Polaroid 49.1 kHz monostatic electrostatic sonar transducer and 6500 ranging module, manufactured by Senscomp, has been the most widely used sensor in previous robotics research [32]. The transducer's beam width is approximately ±15º [33]. Using Equation 3.5, with $\theta = 18º$ or 0.1π, the number of scans, N drops to 20.

The SensComp 6500 ranging module accepts digital initiation, INIT, and blanking inhibit, BINH, signals. It outputs a digital echo signal that goes to a high logic state when an echo is detected. Its range accuracy is a function of the timing accuracy between the time INIT is sent high and the time the echo signal goes to a high logic state. Previous robotics class work at NPS had demonstrated centimeter accuracy using digital timing devices coupled to the SensComp 6500 modules [34]. This range accuracy was superior to that achievable using the SHARP IR rangers' analog outputs.

The researcher did not have a sufficient quantity of transducers and ranging modules to permit mounting eighteen fixed devices around the periphery of the robot. Further, the Creature's components and chassis did not afford enough space to mount so many fixed devices. It was decided to use fewer devices and scan them in azimuth, but this solution created added complexity.

The sonar system consists of four components: a sensor head controller, a servo motor, four ultrasonic transducers, and four ranging modules. The scanning sonar sensor head needed to complete the following tasks:

- mount the sonar transducers and mechanically scan them
- regularly send the INIT pulse to multiple SensComp 6500 ranging modules

- time the interval between INIT pulse and receipt of echo signal, i.e. the TOF
- store TOF data
- communicate TOF data

b. Sonar Mechanical Mounting and Pointing

Four sonar transducers were mounted to a square section of perf board, measuring three inches on a side. Perf board was chosen because of its light weight and pre-drilled holes, which allowed easy alignment of the orthogonally-mounted transducers. To limit rotational inertia of the scanning head, construction emphasized light-weight parts. It was feared that a heavier head with greater rotational inertia would not accelerate and decelerate to a stop precisely enough when the head was scanned at high speed, roughly one degree per millisecond. This would have caused misalignment between the sonar transducer's beam pattern at the time it was fired and its intended ranging sector.

The transducers and perf board were mounted to a Futaba S3003 RC servo, which provides the rotational motion under the command of the sonar sensor head controller. The manufacturer states that the servo requires 0.23 s to move 60º when supplied 4.8V or 0.16 s to move 60º when supplied 6.0V. Equivalently the servo can move at a rate of 261°/s and 375º/s at 4.8 and 6.0 V, respectively. It can generate 3.2 kg-cm torque at 4.8 V [35]. The servo included a circular servo horn, or mount, with the numbered labels at 90º intervals. The servo positioned the flange to its rightmost limit when a 0.30 ms signal was applied to its position signal input. With the servo case aligned with an imaginary Y axis, the Futaba servo's number 1 mount point was observed to move to a position 90º to the right, or aligned with an imaginary X axis. The number 1 mount point's position was observed and recorded for various control pulse widths. The angular position of the number 1 mounting point and the pulse width obeyed the linear relation below, where T is the control pulse width in microseconds, and θ is the angular position in degrees from far rightmost limit of the servo's rotation.

$$T = 300\mu s + \left(10\frac{\mu s}{1^{\circ}}\right)\theta$$

c. Function of PIC Microcontroller and Sonar Circuit

All other tasks listed above were accomplished using a PIC microcontroller mounted to a purpose-built PCB, designed by the researcher using Cadsoft's Eagle circuit layout software. The PCB was manufactured by Advanced Circuits. The PIC16F690's program is explained in chapter 4, and while the construction of the PCB and the sensor head controller is described below. Briefly, the PIC was programmed to send the appropriate servo pulse to the RC servo to position it to the required azimuth. Four discreet digital output pins on the PIC were connected to the INIT signal inputs of the four SensComp 6500 ranging modules. The PIC was programmed to fire a sonar, and count the oscillations of its internal oscillator between the time the INIT signal was sent high and the time the echo signal was observed to go to a high logic state. The four modules' ECHO signals were multiplexed to simplify the PIC's timing operation. The four ECHO signals were combined using an XOR gate, which resulted in a virtual four input OR gate. Thus, when any one ECHO signal went to a high logic state, the combined echo signal input to the PIC also went to a high logic state and provided a signal to cease counting the internal oscillator, stopping the TOF timing.

A MAX232 RS232 level shifting IC was installed in the circuit to shift the PIC's serial data voltages to the RS232 standard. The ranging modules produce noticeable electronic noise on the supply, ground, and echo lines when the transducers are fired [24]. Two 1000 μF capacitors were installed in parallel to reduce noise on the PCB's five-volt supply during sonar firings. Figures 25 and 26 show the circuit schematic for the sensor head controller.

The servo sensor head controller is powered by the electronics power bus and uses five-volt supply. The PIC16F690 and SensComp 6500 ranging modules receive power via the Sonar switch on the Electronics Bus Panel. The RC servo requires five-volt power as well, and is separately switched via the Servo switch on the same panel.

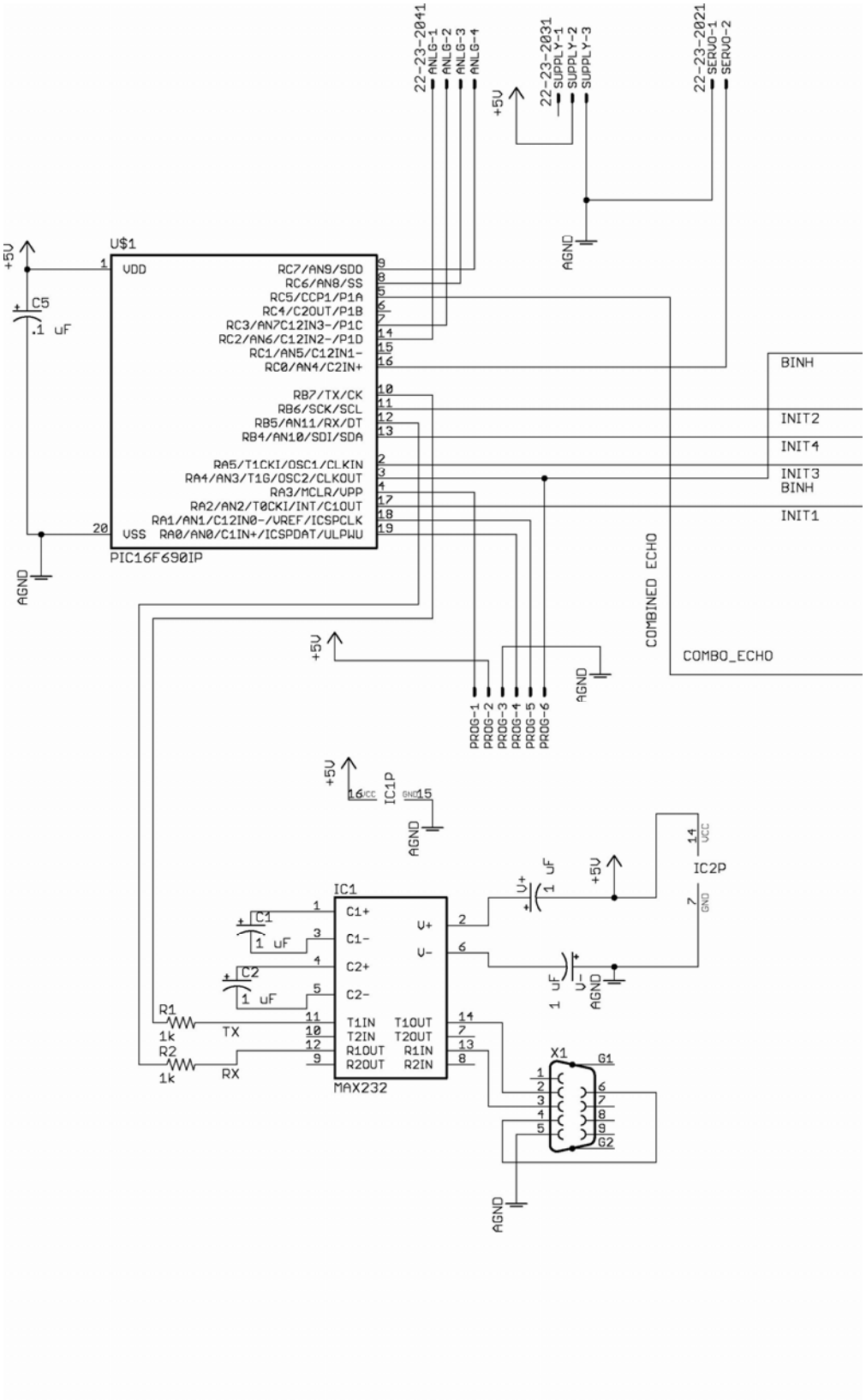

Figure 25. Sonar Sensor Head Controller Circuit.

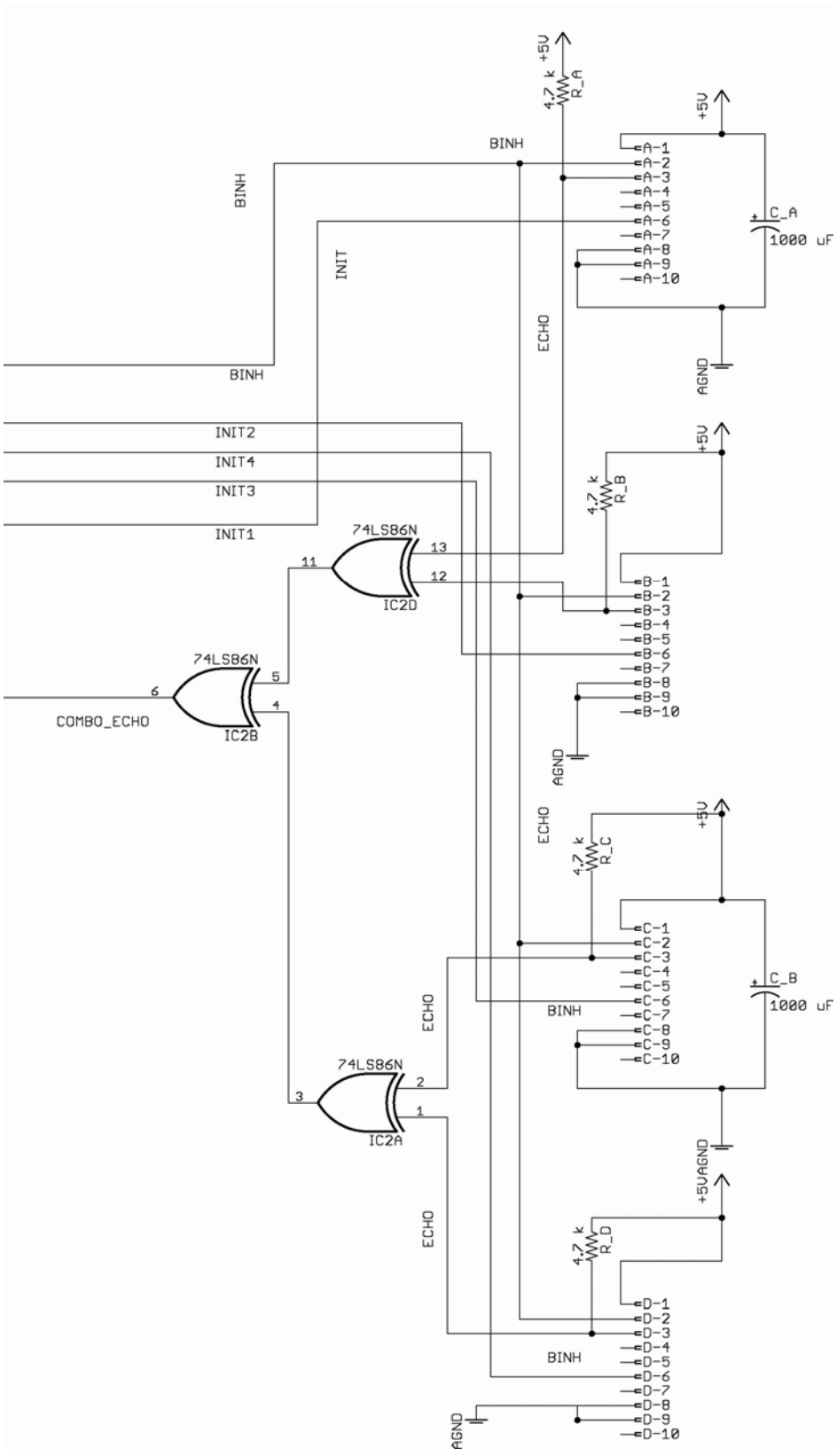

Figure 26. Sonar Sensor Head Controller Circuit Page 2.

G. POSITIONING SENSORS

1. Wheel Tachometers

The wheel tachometer system consists of three binary optical encoder targets, three optical detectors, and a PIC16F690 microcontroller responsible for measuring the frequency of the signals produced by the detectors. Each optical target consists of an aluminum disk screwed into a mounting collar with a hole through its center. The collar was designed to allow the motor shaft to slide through its center. The collar was made to rotate with the motor shaft by two set screws. A Ø 1.75 inch circular target printed on transparency material with 100 sectors, alternating black and white, was affixed to the aluminum disk with glue. There are 50 sectors of each color. The black sectors provide an optical target for a Hamamatsu P5588 photo detector, which produces a logical high signal when a black object is detected. The Hamamatsu photodetectors are designed for applications such as detecting the presence of paper inside a laser printer and are limited to ranges of 1 to 3 mm [36]. Figure 27 shows a schematic of the P5587 sensor, which produces a high output when a light colored target is passed in front. The diagram shows an example circuit and the internal components, including the LED input diode, current amplifier, Schmitt trigger, and output phototransistor, visible. The sensor incorporates built-in hysteresis to ensure the transitions between logic states are free from noise. Its open collector output requires a pull-up resistor R_L to pull the IC's V_o output up to a 5.0 V high logic state.

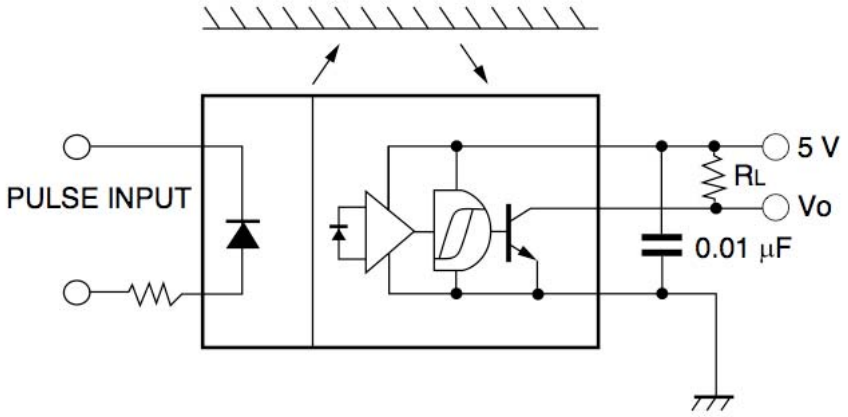

Figure 27. Hamamatsu P5587 photodetector circuit. (From Hamamatsu)

As the motor shaft rotates so does the circular target, passing a continuous series of alternating black and white sectors in front of the Hamamatsu detector. In the P5588 as the edge of a black to white sector passes the detector, the reflected light off the target is detected and the current amplified. On a high to low transition, if the detector's forward current exceeds 10 mA, the high logic state is triggered and the transistor is switched allowing the optical detector's output to be pulled up to 5.0 V. The effect of passing a repeating series of alternating sectors is a clock signal with frequency proportional to the motor shaft speed. With a 100 sector optical target installed the Hamamatsu sensor generated a clock signal with 50 cycles per shaft revolution, e.g. 50 cycles per two π radians angular rotation.

All three motor shafts have a target disk mounted to them. Three compact PCBs were built to mount the detector, pull-up resistor, and noise capacitor. These PCBs were located inside the three motor mounts. Figure 28 shows a photograph of one optical target and its detector inside a motor mount box. The detector PCB is on the right, and one can see the Hamamatsu photodetector slightly to the right of the center where it is mounted in close proximity to the target disk, approximately 1 mm.

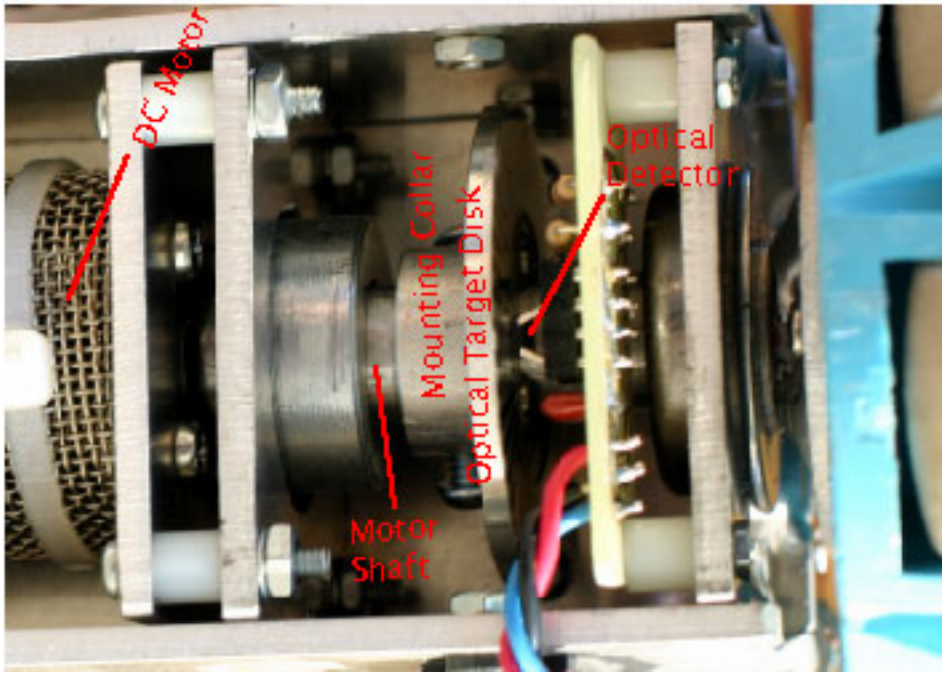

Figure 28. Wheel speed optical target disk and detector.

By loosening the collar's set screws, one can adjust the position of the target along the shaft's axial direction until it is detectable by the Hamamatsu sensor, at which point the set screws can be tightened. It was observed that care must be taken when tightening the set screws to prevent the target disk from being tilted. If the target disk "walks," then it will no longer be normal to the optical detector, and portions of the disk will be farther from the Hamamatsu detector when they pass its FOV. Some sectors were observed to fail to register due to the target's excessive distance from the detector. If the target is placed too close to its detector, the two can rub and result in a damaged target. The effect is a target with a "dead" section where no clock signals are generated.

2. Inertial Measurement Unit (IMU)

The Creature is the first SMART program robot to mount an IMU. Because the intended operating environment would not allow for reliable receipt of GPS signals, it was decided to incorporate an IMU to research inertial navigation. The Onavi Falcon GX was selected based on its cost, ease of integration with the existing robot CPU via RS-232, and its features. The Falcon incorporates three orthogonally mounted linear

accelerometers and three angular rate sensors. The combined output is continually reported via RS-232 in either ASCII or binary format. Immediately after power up, the FalconGX allows the user to configure its RS-232 baud rate, and the data update rate. The default RS-232 data rate is 9600 b/s, and the update rate 1Hz [37]. To decrease the time required for receipt of the complete IMU data sentence, the RS-232 rate was changed to 19.2 kb/s. The update rate was set to 20Hz in order to provide the BL2600 with timely rotation rate data when turning, since the Creature was observed to be capable of rotation in place at rates exceeding 180º/s.

The IMU was mounted to an aluminum plate that provided added mass and served to damp out small amplitude vibrations. The plate and IMU were attached with Velcro to the chassis in the geometric center of the robot. The IMU was installed such that its positive X, Y, and Z axes were aligned with the robot's X, Y, and Z axes, respectively. Velcro allowed for easy adjustment of the device to align it with marking on the chassis that indicated the direction of the robot's Y axis. The IMU's Z axis coincides with the robot's Z axis, but according to the manufacturer, the IMU's positive z rotation is opposite to the sign convention used in the robot's frame of reference, i.e. positive z rotation about the IMU's Z axis is clockwise as viewed from above and would be considered negative rotation about the robot's Z axis in the robot's frame of reference. Figure 29 shows an illustration of the orientation of the IMU's linear accelerometer axes and rotation sense of the rotation rate sensors. Note, that positive IMU rotation rate data about its Z axis corresponds to negative rotation in the Creature's frame of reference introduced in Chapter II.

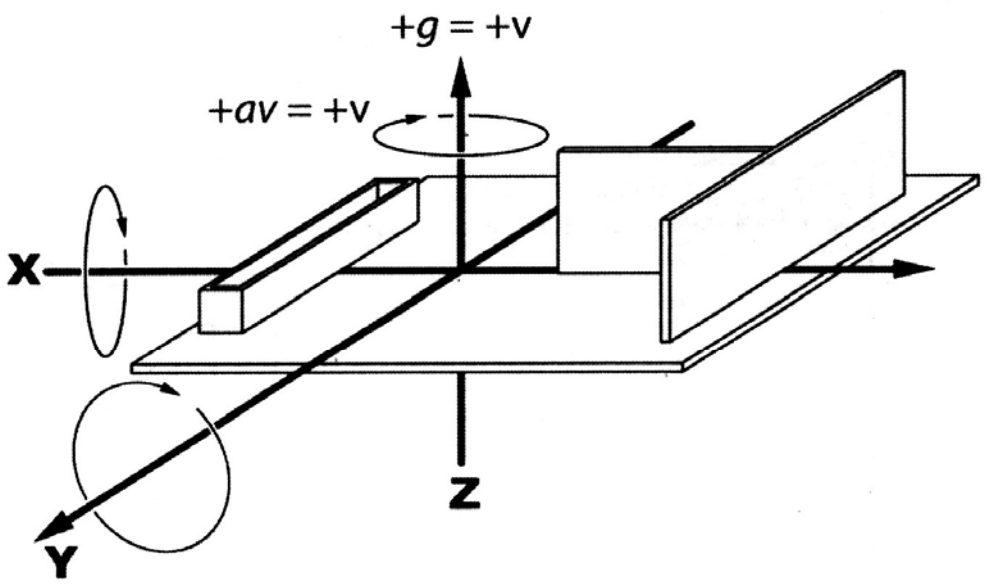

Figure 29. FalconGX IMU Axes. (From ONavi)

The IMU's linear accelerometer and rotation rate sensors are analog devices whose output is converted to 10-bit digital representation by built-in ADC. The 10-bit data scheme assigns zero g, or 0.0 m/s^2, to a value midway between zero and 1023, i.e., equal to 511. Stationary in the lab, the IMU reported -1 g, or - 9.80 m/s^2, acceleration along the Z axis,. The reported data is a 10-bit representation of the IMU's analog device readings, and it must be scaled appropriately if one desires MKS units. The equation below gives the necessary conversions for the reported linear acceleration, a 10-bit representation of the device's full scale 4g range, to acceleration in MKS. Rotation rate must be scaled as well using the full-scale value of 300º/s [37].

$$acceleration[m/s^2] = \left(\frac{reported\ value - 511}{1024}\right) 4g \left(\frac{9.80\ m/s^2}{g}\right)$$

$$rotation\ rate[rad/s^2] = \left(\frac{reported\ value - 511}{1024}\right) \frac{300^\circ}{s} \left(\frac{2\pi\ rad}{360^\circ}\right)$$

Data are reported to the BL2600 in ASCII format, which allows easier interpretation by humans. It was decided to use this format during integration and testing, despite the fact that binary communications would decrease the total bytes, and thereby the time, needed to transmit the IMU's data sentence.

IV. SONAR SENSOR HEAD CONTROLLER MICROPROCESSOR ASSEMBLY LANGUAGE PROGRAM

A. PROGRAM TASKS

The hardware components associated with the sonar sensor head controller have been discussed previously. To provide flexibility and reduce hardware component count a microprocessor solution was developed to trigger the sonar ranging modules and measure the TOF. This section describes the PIC16F690 microprocessor's assembly language program. The microprocessor is responsible for the following tasks:

- Sending individual INIT pulses to different SensComp 6500 ranging modules
- Timing the interval between INIT pulse and receipt of echo signal, i.e. the TOF
- Storing unique TOF data for each sector around the robot
- Communicating stored TOF data to BL2600

B. SONAR CIRCUIT AND PROCESSOR TIMING EXPLANATION

A timing diagram for two of the four sonar channels is provided in Figure 30. It shows two sequential sonar ranging attempts. In the first ranging attempt, the number four SensComp 6500 ranging module sets its ECHO signal high after detecting a valid return echo. In the second ranging attempt, the number three ranging module does not detect a valid return echo, so its ECHO signal remains low. In the first case the sonar sensor head controller PIC saves the value of the sixteen-bit counter when the combined echo transitions from a low to high logic state. After the value is saved, the timer is allowed to overflow to ensure the period between RC servo pulses is approximately 30 ms. In the second, no echo is detected, so the sixteen-bit timer's value reaches its maximum, and the timer overflows. The PIC treats this as a non-detection in its code.

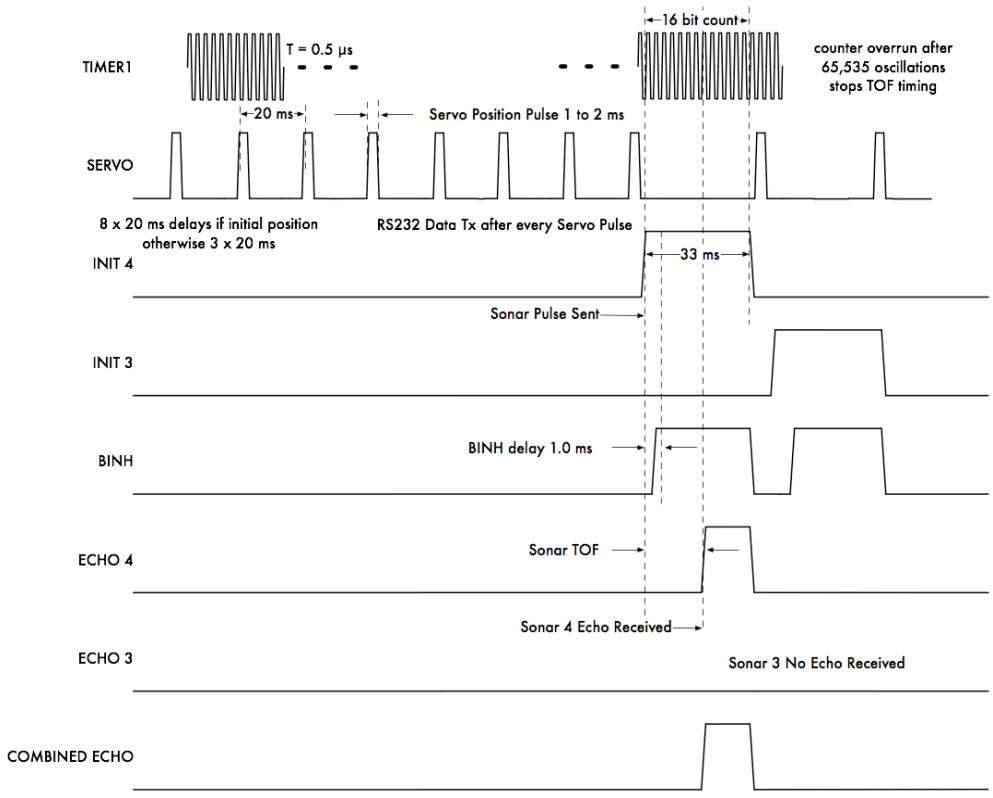

Figure 30. Sonar Controller Timing Diagram.

C. PROGRAM FLOW

The program begins by configuring various registers. The PIC is configured to use its internal RC oscillator for the clock source, the code sets its speed to 8 MHz. It configures all inputs to digital and the appropriate inputs and outputs per the table above. TIMER1 is configured to increment at a 1:1 ratio with the instruction clock, which is one quarter of the processor frequency, 8 MHz. Thus, TIMER1 increments every 0.5 µs. The RS232 UART is configured for asynchronous serial operation at 57.14 kbit/s, eight-bit data, with no parity. Figure 31 shows a flow chart that describes the microprocessor's program. The program's flow chart is continued on Figure 32.

The program uses a 20 to 35 ms time period as the primary building block for all activities because the RC servo must receive its positioning signal at an interval on the order of 20 to 40 ms. After configuration and initialization, all program actions take

place within a regular 20 to 40 ms period or time block. The main loop begins by moving the servo sensor head to its initial position. The subroutine SubSendServoPulse is called to send the servo a high logic pulse. The pulse's duration is proportional to the angular position desired. The subroutine uses the value of the position variable in a case-switch statement to choose the length of the high pulse. The RC servo signal must be repeated after roughly 20 ms, and the program calls the subDEL20 subroutine to produce the delay. If the position is the initial position, then a longer delay consisting of eight iterations of the servo pulse and 20 ms delay is provided in the program to allow the detector head extra time to move from its final position to the initial position, an angle of 72º. Based on the Futaba S2003 specification rotation rate of 261º/s, one would expect the rotation from the last position to the starting, initial position to require 276 ms [35]. When the servo is only required to move 18º between sonar firings, the servo pulse and 20 ms delay is repeated just three times because the smaller angular displacement requires less time for the servo to complete.

After the servo moves to the correct angular position, the program commands a sonar ranging. Each of the four sonar transducers is triggered sequentially beginning with sonar number four. The appropriate INIT signal output is taken to a high logic state, causing the corresponding SensComp 6500 ranging module to trigger its sonar ranging cycle, and the transducer transmits sixteen pulses. The PIC's enhanced capture and compare module is configured to trigger on the rising edge of the combined ECHO signal and the associated interrupt flags are cleared. Then TIMER1's count is cleared and the counter started.

The SensComp 6500 module has a default 2.38 ms delay after transmitting the sixteen sonar pulses to prevent confusing the transducer's ring down as a return echo [24]. Assuming the speed of sound in air equals 343 m/s, this limits the minimum range to 80 cm. It was decided that this minimum range was unacceptably large for the expected operating environment. At the risk of detecting the ring down as echo the PIC overrides the default echo blanking by sending the blanking inhibit BINH to a high logic state after 1 ms. The 1 ms BINH delay has been observed to provide a 20 cm minimum range.

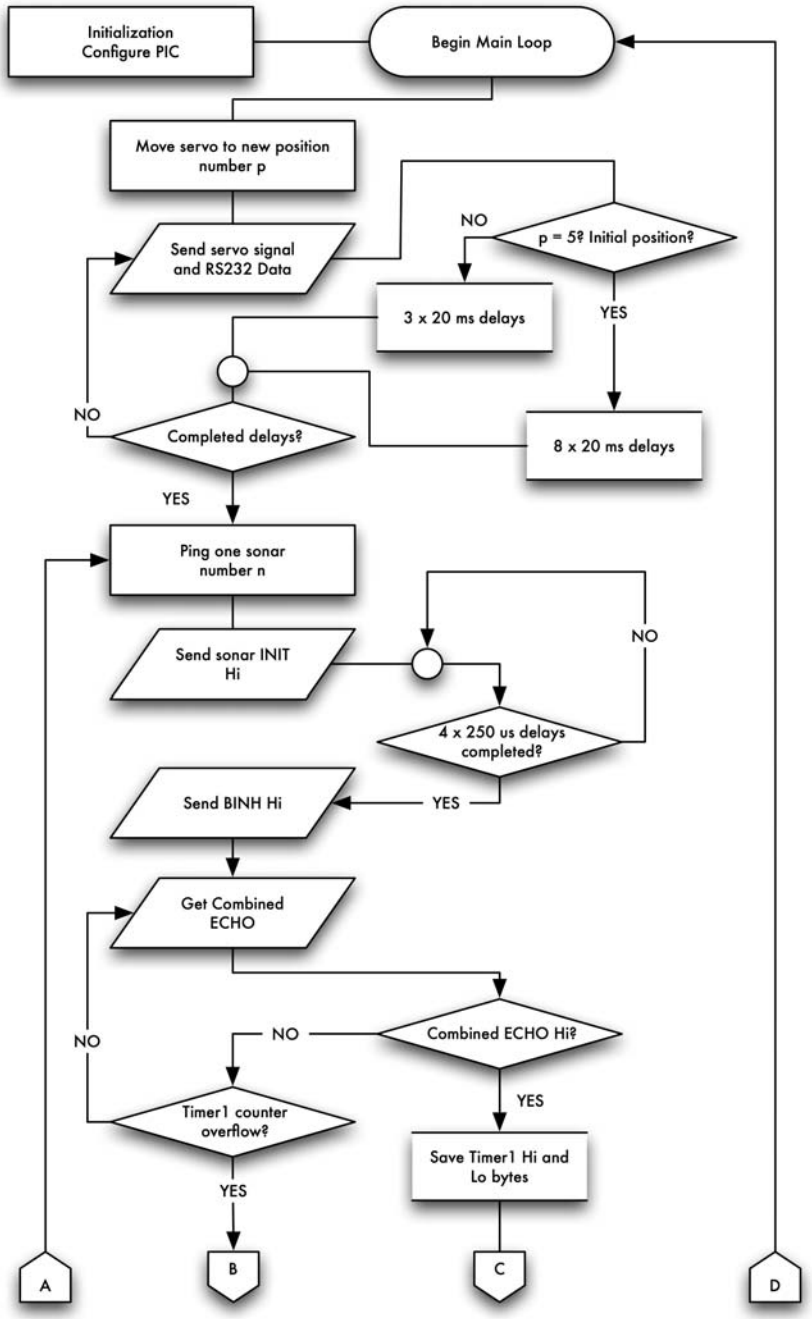

Figure 31. PIC16F690 Sonar Sensor Head Controller Program Flowchart.

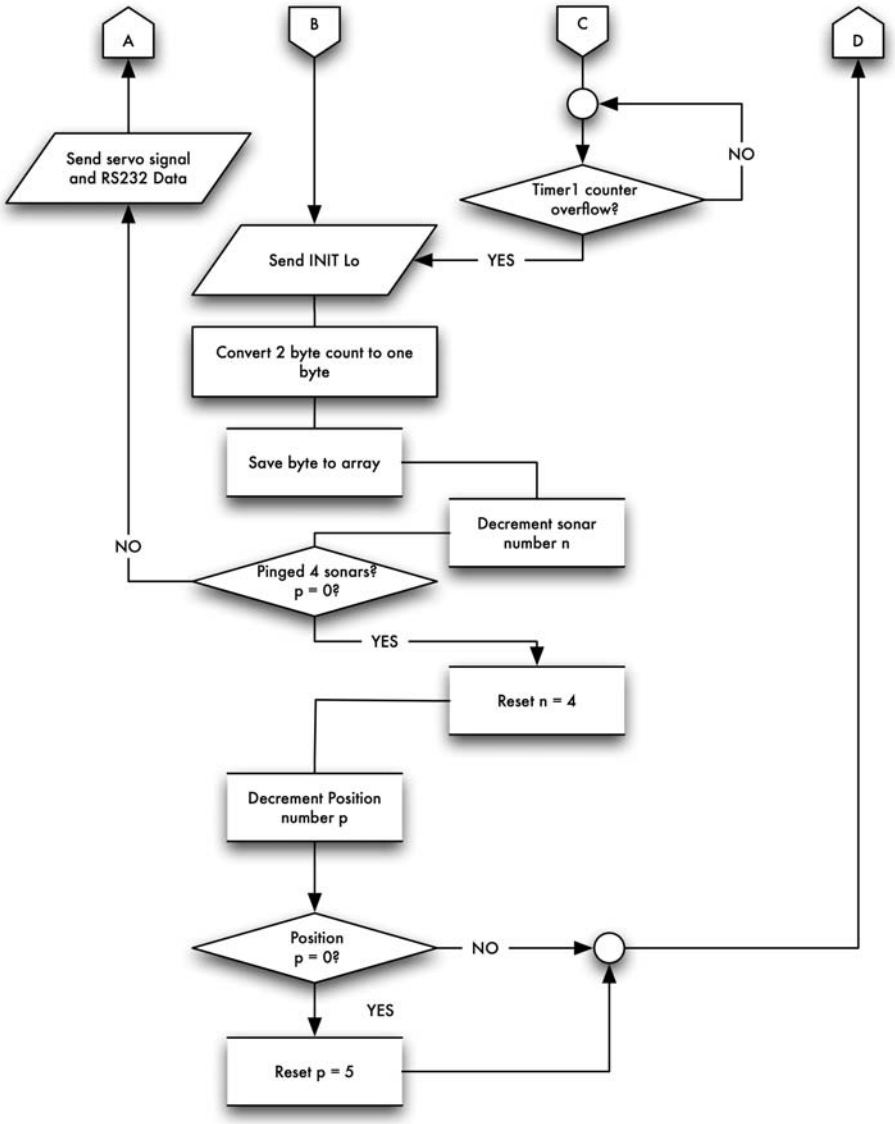

Figure 32. Continuation of Program Flowchart.

Then the PIC begins a loop that checks the interrupt flags for either receipt of an echo or a TIMER1 counter overrun. If the SensComp 6500 ranging module detects a valid echo, it sends its ECHO signal high. The four module's ECHO signals are combined via a XOR gate. If the combined echo transitions from a low to high logic state, both of TIMER1's counter bytes are saved, and then TIMER1 is allowed to finish

counting up until overflow. This ensures the process' duration takes 33 ms regardless of the length of time between sonar firing and reception of the echo, and it guarantees the RC servo pulse will be sent again after an appropriate delay.

The most significant byte is tested to determine if the counter value indicates receipt of an echo from a target at a range that exceeds the limits of the reportable range. These TOF values are set to a literal value 0x02 to flag the data as no contact. The most significant byte and least significant byte of the counter variables are combined to make a one-byte value of the count. The 2^7 bit through the 2^{14} are retained, and the byte is saved to an array whose index corresponds uniquely to the combination of sonar number and servo position using the relation below.

$$array\ index = 4(5 - position) + (4 - sonar\ number)$$

The variable SonarNum is decremented to allow the PIC to complete a sonar ranging with each of the sonar modules from four to one. The inner loop is repeated. If decrementing the variable results in a value of zero, then all sonars have been fired and the position variable is decremented. The position loop is repeated until decrementing the position variable results in a value equal to zero. Then the position value is reset to its initial value, five, and the outermost loop begins again with the sensor head being commanded to move to the initial position.

D. PROGRAM OUTPUT

The PIC transmits the one-byte counter representation of the TOF through its built-in UART. To ensure the robot's BL2600 CPU is not delayed any longer than necessary, the PIC transmits all data available as part of the subroutine that is responsible for sending the RC servo pulse. Rapidly refreshing the data also ensures that the CPU is afforded many opportunities to receive the range data. There is no need for the CPU to interrupt its processing to read in the PIC's transmission. The CPU can expect to receive the 20 range data bytes at approximately 20 to 33 ms intervals. The communicated sentence uses flags with hexadecimal values that are outside the counter values for

minimum and maximum counts. Since the reported range is limited to 250 maximum, values 251 to 255 are available for the program's use as flags. The minimum range should correspond to a count of 19 or 20, which allows values 0 to 18 for use as sentence flags. The hexadecimal value 0x02 indicates no echo was received or that the echo was detected beyond roughly 2.5 m. The values 0xFE and 0xFB indicate sentence start and end, respectively.

Figure 33 shows an example data sentence from the sonar controller. The sentence is ordered such that the first range byte corresponds to the sector aligned approximately with the robot's Y axis. Moving clockwise around the robot, succeeding bytes represent the range for the next eighteen-degree sector. Thus the tenth data byte corresponds to the sector roughly aligned with the robot's negative Y axis. After the twentieth data byte is sent, the PIC transmits an integer value between 0 and 19 to indicate which array value is most recent.

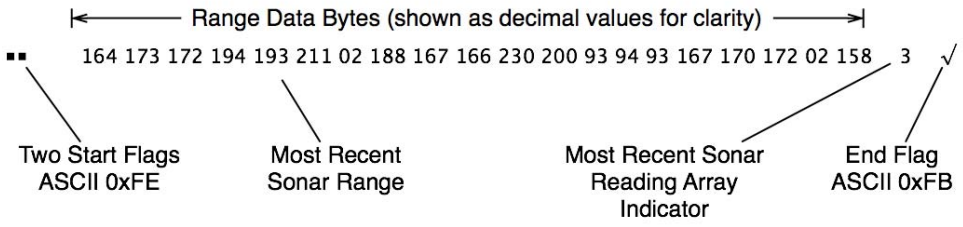

Figure 33. Sonar Range Data Sentence.

Note the integer value uses the PIC's indexing scheme, which corresponds to the sequence that the sonars were fired. The PIC stores the data bytes with an array index according to the order the sonars ranging attempts were made. Because the twenty azimuth scans are interleaved, the PIC's internal data array index does not correspond to the same clockwise azimuth order that is reported. Translating between the two indices is accomplished with a simple lookup table, and one can determine how time-late each range is for any sector.

Sonar Sentence Order	Azimuth (relative to Y axis)	PIC Data Array Index
0	6º	2
1	24º	19
2	42º	15
3	60º	11
4	78º	7
5	96º	3
6	114º	16
7	132º	12
8	150º	8
9	168º	4
10	186º	0
11	204º	17
12	222º	13
13	240º	9
14	258º	5
15	276º	1
16	294º	18
17	312º	14
18	330º	10
19	348º	6

Table 2. Sonar Data Array Index to Azimuth Cross-reference.

If one assumes the robot travels in a straight line without rotating between received data sentences, then older range readings can be adjusted for robot motion by multiplying the robot's velocity vector times the time since the sonar detection was received. The effect is to move all range readings to objects, except the current one, by a distance proportional to the apparent motion between the robot and the object.

The following table lists the connections used by the PIC in addition to the In-Circuit Serial Programming (ICSP) pins.

Description	PIC Connection	IC Pin Number
RC Servo Position	RC0	16
Sonar 1 INIT	RA2	17
Sonar 2 INIT	RB6	11
Sonar 3 INIT	RA5	2
Sonar 4 INIT	RB4	13
Blanking Inhibit (BINH)	RA4	3
Combined ECHO	RC5	5
RS232 TX	TX	10
RS232 RX	RX	12
+5 V	VDD	1
GND	VSS	20

Table 3. PIC Connections to Sonar Circuit.

THIS PAGE INTENTIONALLY LEFT BLANK

V. THREE WHEEL TACHOMETER MICROPROCESSOR ASSEMBLY LANGUAGE PROGRAM

A. DESIGN CONSIDERATIONS AND PREVIOUS WORK

The functionality and flexibility of the PIC16F690 microcontroller prompted the researcher to develop a wheel tachometer based solely on it. Until the Creature, SMART program robots had not incorporated wheel speed sensors, nor been equipped to perform odometry calculations [10, 11]. Based on previous NPS robotics class work, the researcher was familiar with binary encoders and odometry using optical targets and detectors. A hardware solution had been built, tested, and demonstrated on a chassis similar to Herkamp's, but this device was extremely restricted in its usefulness and used a very rudimentary three wire serial communications scheme to "bit bang" its data to a BL2000 [34]. Further, the device attempted to measure distance directly through a count of the encoder's rotation.

B. TACHOMETER PROBLEM SOLUTION

A better solution needed to be found. Using the same type of optical targets and sensors, a more robust and flexible solution was devised using a single PIC16F690 in place of the PCB described above, which included seven ICs. The PIC allowed the researcher to use common I2C bus communications and added unparalleled flexibility through implementation of a software solution. With a software-based PIC solution, the tachometer can easily be modified to operate on other robots with two, three, or four odometry sensors. Further, the actual sensor is immaterial; the PIC can be adapted to work with any regularly occurring digital signal such as one from a Hall Effect sensor. The PIC sensor could be used to provide feedback about any motor shaft motion, such as motors used to position a robotic arm. This adaptability was a design goal throughout the development. Solutions should be flexible enough to allow for their implementation on follow-on SMART program robots.

Whereas the earlier device used hardware to measure displacement, an angular velocity measurement technique was selected for the Creature to smooth out the observed data. The microcontroller's fundamental task is to count the number of cycles reported by each sensor in a fixed period. The PIC16F690 used for wheel speed measurement accepts three inputs, one from each Hamamatsu optical sensor observing the Creature's three motor output shafts. The sensors provide a digital signal that goes to a high logic state when the black portion of the binary encoder wheel passes through its FOV. The PIC's program counts the number of transitions, i.e. changes from high to low or low to high logic states, in a fixed period of time. If one assumes that one cycle includes two transitions, dividing the count of the transitions by two times the period gives the frequency in cycles per second of the motor shaft. A 250 ms period was chosen as the sampling period. A longer period would have smoothed the frequency data more and mitigated the observed optical sensor errors due to occasional failures to detect some target sectors. Additionally, detecting low frequency signals (10 to 17 Hz) associated with very slow wheel speeds required a longer sampling time. Longer periods, though, obscured short-term variations in motor frequency, for example during accelerations. Ultimately, a 250 ms period was chosen as a compromise. Further, it allowed for a one-byte data value at the maximum expected signal frequency of 220 to 250 Hz, and simplified the assembly language mathematics, since multiplication by two consists of a simple bit shift.

In addition to measuring the motor shaft frequencies, the PIC must also store the data and communicate it to the robot's CPU, the BL2600. It was decided to use I2C bus communications for this application because the CPU's serial ports were committed to sonar and IMU data communications. Rudimentary communications protocols outside the mainstream have been used on some earlier odometry sensors, but such protocols violated the design goal that devices built for the Creature be adaptable to a wider range of vehicles. Using the I2C bus protocol allowed the CPU, serving as bus master, to poll slave devices at its convenience. By using the I2C protocol the data was available to the CPU with minimal delay. I2C uses a simple bi-directional, two-wire hardware layer that was already installed, and the CPU's Dynamic C language provides a feature-rich I2C

function library. Previous SMART robots, such as Herkamp's Bigfoot, have utilized I2C communications, but the bi-directional nature of the signals coupled with I/O limitations of the BL2000 forced the researcher to divide the SCL and SDA lines into an input and an output pin for each signal [11]. The BL2600 installed on the Creature was configured to use sinking current outputs, and achieves true bi-directional serial communications through two pins.

C. PROGRAM FLOW

Figures 34, 35, and 36 show flow charts of the wheel tachometer assembly language program. The main program element is a loop during which the PIC monitors three digital inputs. If a transition, a change in the digital input from a low to high logic state or high to low logic state, is detected, then the register associated with that motor shaft count is incremented. A short 40 μs delay is implemented after the three signals are tested for a transition in their logic states to allow for rise time in the digital signal lines. The PIC's TIMER1 counter is seeded with a literal value prior to executing the loop. During code execution, the TIMER1 counter increments automatically from the seed value until the counter overflows, which occurs after 250 ms have elapsed. The overflow sets an interrupt flag.

The remainder of the program, shown in Figure 36, is an interrupt service routine (ISR) handler. Because the researcher lacked extensive assembly language programming experience, example code to handle the ISR tasks and I2C communications was found [38]. The example was modified to measure the wheel frequencies. The code monitors for ISR triggered by the overflow of the counter, which signals the completion of the 250 ms timing. If the ISR was caused by the counter overflow, the counter's overflow flag is reset along with the seed value for the counter for the next 250 ms period. To produce an output value in cycles per second from the number of transitions in 0.25 s, and assuming one cycle per two transitions, the code multiplies the number of transitions by two. The three values representing the frequency in cycles per second for the wheel rotational speed are saved to separate registers so as to be available when requested by the I2C bus master.

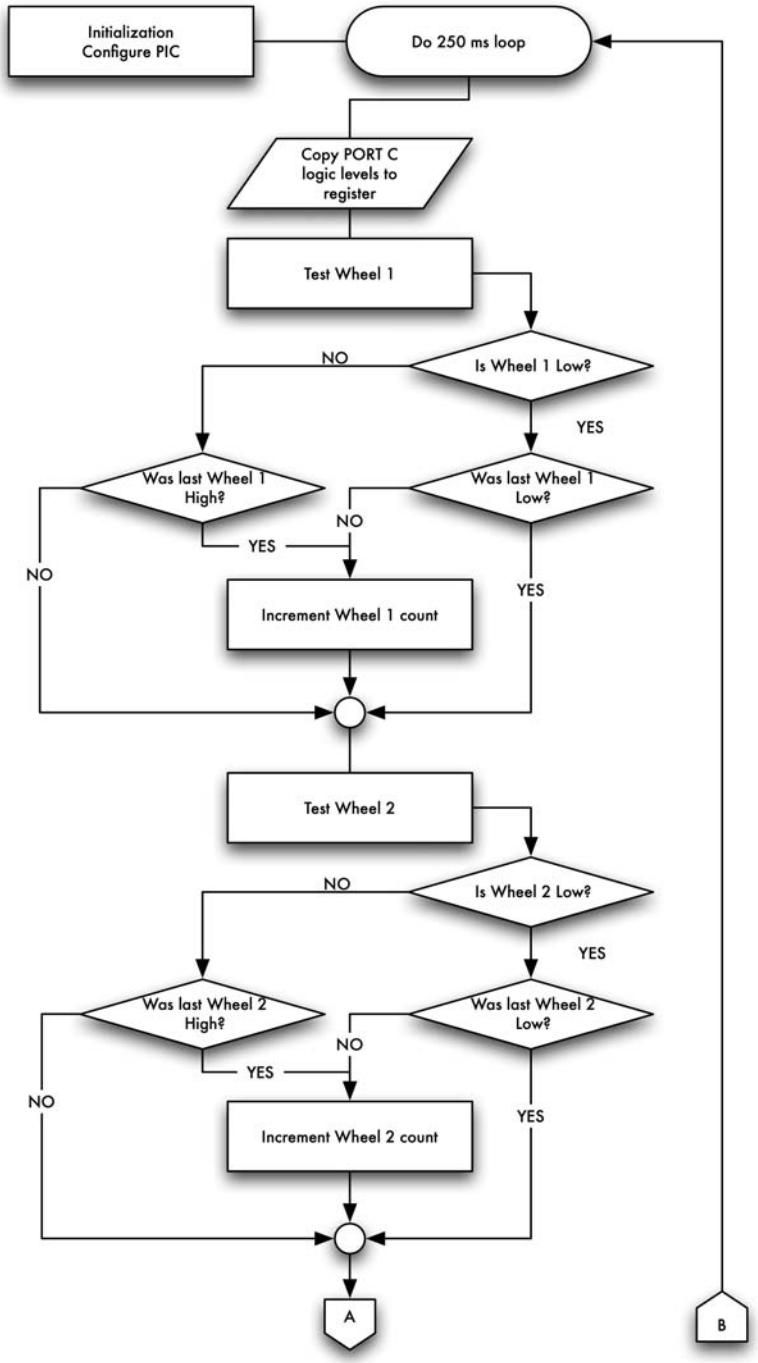

Figure 34. Tachometer Program Flowchart Page 1.

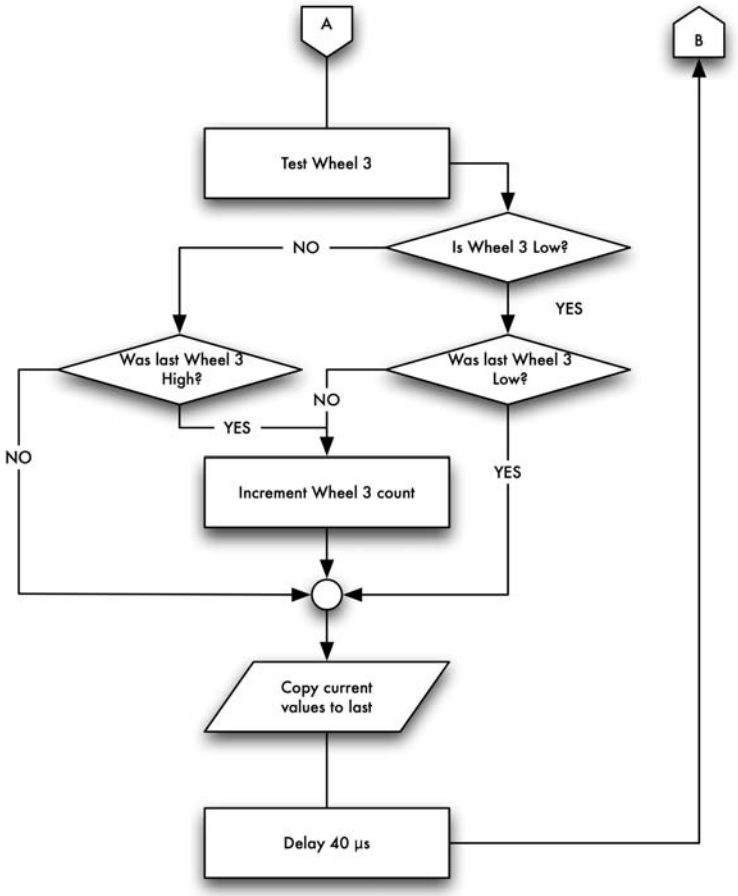

Figure 35. Tachometer Program Flowchart Page 2.

D. DATA OUTPUT

The ISR hander also parses ISRs triggered by I2C communications. A complete I2C bus communications description is beyond the scope of this thesis. One should consult Philips Semiconductor's comprehensive document for more information [39].

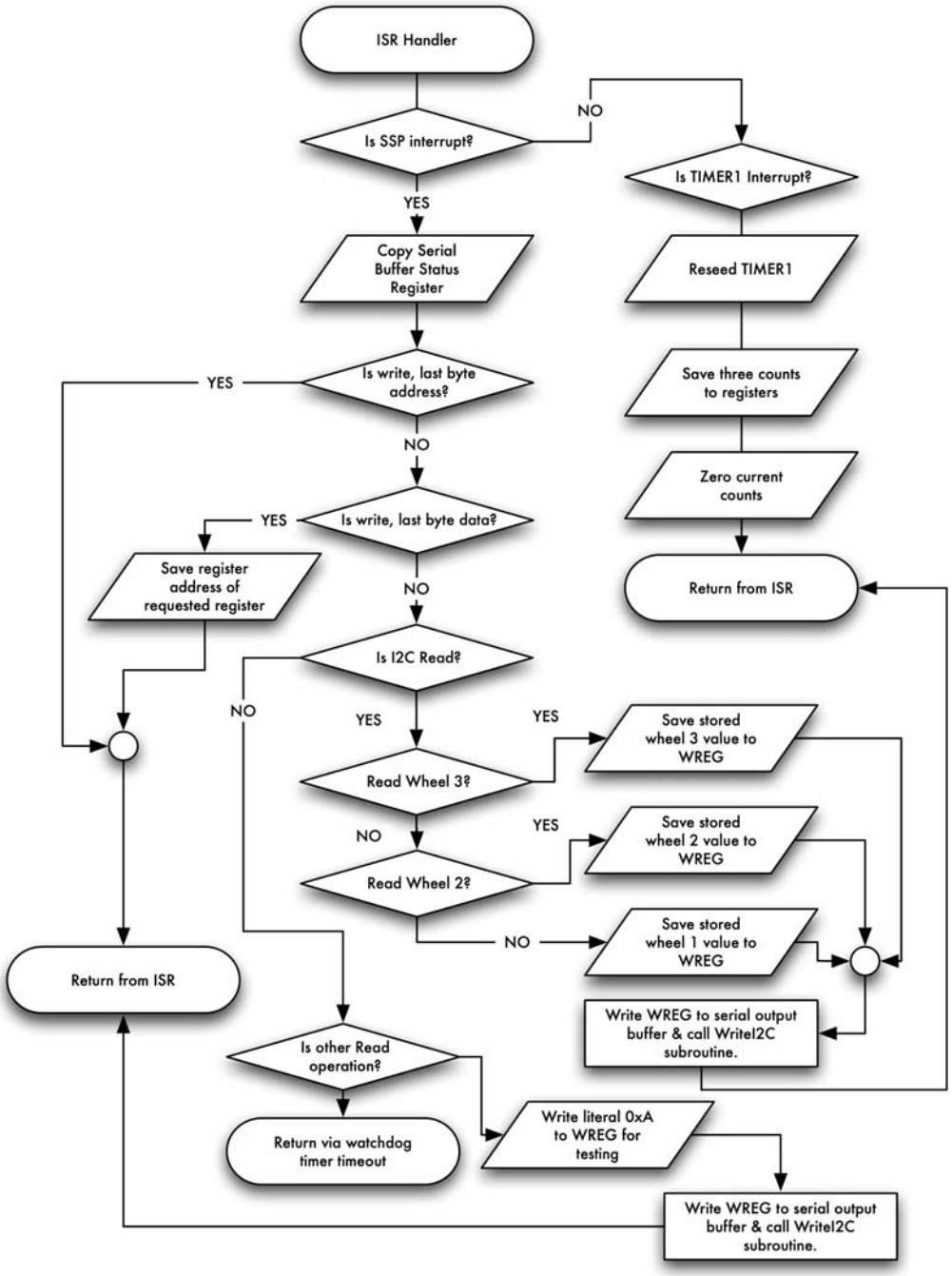

Figure 36. Tachometer Program Flowchart Page 3.

Briefly, the BL2600 serves as the I2C bus master and controls the serial clock line, SCL. It initiates all communications at its leisure. To request wheel tachometer data from the PIC serving as the tachometer the BL2600 sends a start sequence, then the

tachometer's seven-bit address, 0xD0, with the LSB clear. The PIC's built-in synchronous serial port (SSP) module detects the I2C start sequence and verifies that the address matches the seven-bit address set in the configuration portion of the code. If the address matches, the PIC's serial port triggers an interrupt condition that can be parsed in the ISR hander. Next the BL2600 sends the byte of the register whose data it is requesting to read, and then resends the start sequence. Each event triggers an ISR and the ISR handler calls the serial port handler, SSP_Handler. In the serial port handler, the serial port's status flags are sequentially evaluated to determine which state of the I2C communications sequence is occurring, i.e. start sequence, write to PIC, read from PIC, etc. In State 3 of the serial port handler, the requested register byte sent from the BL2600 is tested to determine which wheel speed value was requested. The appropriate register corresponding to one of the three wheel speeds is saved to the PIC's working register, WREG, in anticipation of the BL2600's command to transmit data. In State 4 of the I2C communication sequence, the BL2600 sends the PIC's seven-bit address with the LSB set to command the PIC to transmit its data. The PIC responds by calling the WriteI2C subroutine to write the value of the wheel speed in cycles per second, to the serial port's buffer for transmit to the BL2600.

The BL2600 Dynamic C code provides a feature-rich set of functions to communicate with I2C devices. Although the I2C specification provides for data rates to 400 kbit/s, the Creature's I2C communications were limited to a maximum 100 kbit/s due to the published maximum data rate by the Devantech digital compass [40]. Operation at 100 kbit/s was not deemed possible due to the BL2600's inability to provide for precise timing delays below 1 ms and for fear of introducing errors due to missed bits. In place of timing delays using Dynamic C's millisecond timer function, the BL2600's I2C code was edited to use a short for loop to count from zero to four between I2C clock pulses.

Observations of the SCL and SDA serial data transfers were made using an oscilloscope. The for loop delays the processor approximately 25 μs between I2C SCL clock pulses, which provides a 20 kbit/s data rate. A complete I2C sequence to request one byte of data and read it in from a slave device, i.e., the Devantech CMPS03 magnetic compass, was observed to take 3.4 ms. This value is approximately three times longer

than predicted based on the assumed number of bits that must be clocked out by the master to communicate the data between the devices. The BL2600's multithreading operation is the likely cause of the increased time required, since the processor cannot be guaranteed to execute code sequentially.

VI. ROBOT OPERATING PROGRAM

The Creature's operating program is written in Z-World's proprietary Dynamic C 9.21 programming language, which resembles the C language, and runs on a BL2600 SBC. Like previous SMART program vehicles, the program was edited and compiled using the associated integrated development environment (IDE). The operating program is included in Appendix F.

A. CREATURE'S OPERATING PROGRAM REDESIGN MOTIVATION

A number of robots preceding the Creature have used the Dynamic C language as the basis for their operating programs. Miller employed an earlier Dynamic C version, 7.04, with the BL2000 SBC [10]. Herkamp and Jun's Bigfoot robot used the combination of Dynamic C 9.21 and the Z-world BL2000 SBC with success [11, 28]. The monolithic program used by Miller was an outgrowth of earlier work on the Bender robot, while Herkamp's use of version 9.21 of Dynamic C represented a new line of operating program development [10, 11]. A complete description of the prior operating programs' organizations is beyond the scope of this thesis. Prior operating programs displayed severe time lags during their program loops, though. Herkamp noted the Bigfoot operating program required 665 ms to complete its I2C communications with the CMPS03 digital compass, and Bigfoot occasionally failed to react in time to avoid collision with obstacles [11].

B. OPERATING PROGRAM IMPROVEMENTS

The researcher decided that a design goal for the Creature should be computational load distribution. In effect, it was decided to eliminate low-level tasks from the CPU's robot operating program to free its processor for higher-level tasks to reduce the robot operating program's loop cycle time. The sensor section describes how PIC microcontrollers were used towards this end. Computational load distribution was merely the first step in improving program execution speed. The Bigfoot Dynamic C code was abandoned. Code from previous NPS robotics class work by Le, Cole, and Gamble provided a starting point, instead [41]. Its TCP/IP functions had been

successfully demonstrated in class work, and showed promising improvements, namely reduced communications latency. It featured a modular, procedural structure with execution branching off to complete a variety of tasks handled by discreet functions.

The Creature's operating program substituted Dynamic C library functions for I2C communications. Inefficient, CPU-intensive delays using loop counters were eliminated. Expected delays between the CPU and peripheral devices such as sensors, were accommodated with Dynamic C's built-in multithreading costatements. Code associated with equipment that was not installed, such as a Global Positioning System (GPS) receiver, was deleted. Much new code was created to interface with the newly created sonar sensor and three-wheel tachometer and to operate on the received data. IMU code was developed. Directional control feedback was added using the IMU rotational rate data about its Z axis. Navigation was approximated using a flat-Earth in the vicinity of the robot, and navigation from wheel odometry was added.

C. OPERATING PROGRAM FLOW

The program begins by declaring variables whose scope is limited to the main function. Then the BL2600's digital and analog inputs and outputs (I/O) are configured followed by a function call to initialize the robot, which effectively sets various global variable default values and sets up serial communications between the CPU and the sonar and IMU. The operating program then proceeds to enter its main loop, which comprises the sixteen concurrent costatements shown in Figure 37. Using the multithreaded nature of Dynamic C's execution of costates, the program employs shorter or longer time delays before execution is allowed to return to a particular costate. High priority tasks use a short time delay between costatement revisits, while lower priority tasks give up execution for longer periods.

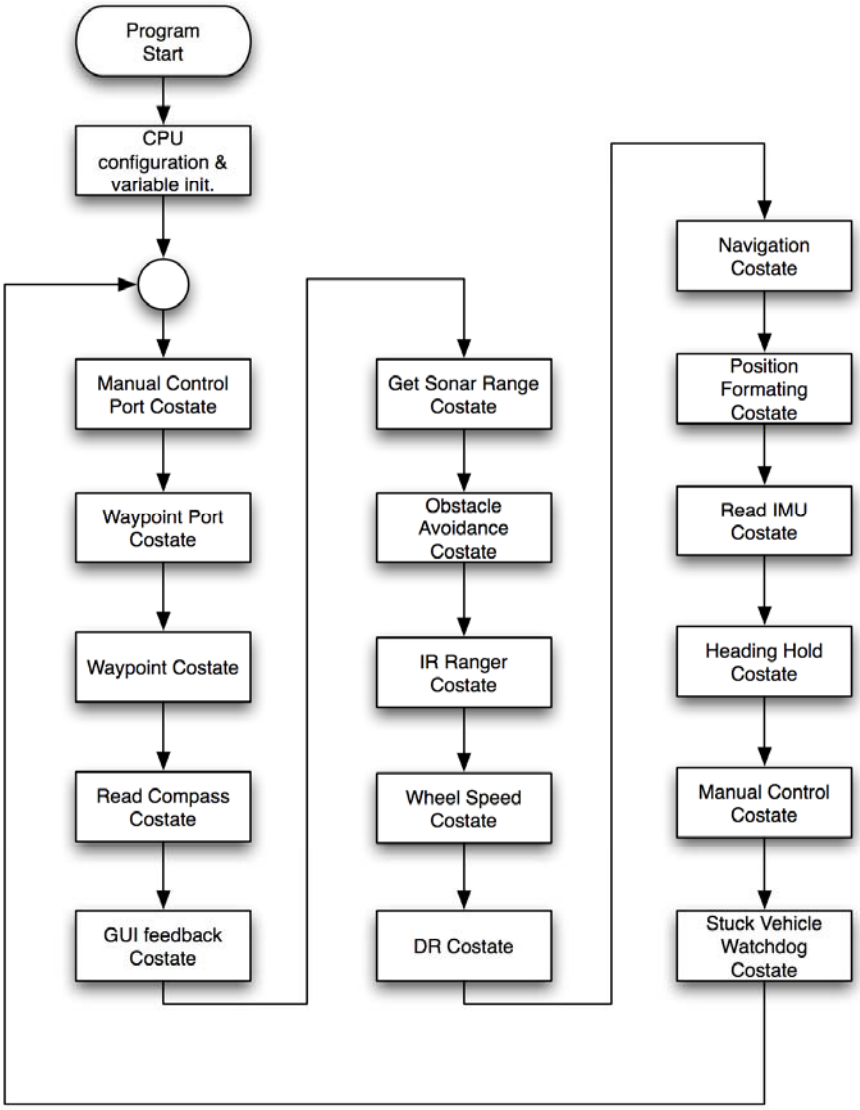

Figure 37. Flowchart of Robot Operating Program.

1. Port Checking and Waypoint Costates

The first two costatements in Figure 37 check for the presence of TCP/IP data packets on their ports. They are responsible for receiving the TCP/IP packets through

function calls to receive_packet_from_port. The Waypoint costate is triggered if waypoint data is received by the above costate. It saves the waypoints transmitted by the laptop's Java graphical user interface (GUI) and calls functions to convert and save the waypoint data into a Cartesian format. Next it initializes navigation and saves a waypoint array before starting the robot's autonomous operation, setting its direction to drive, setting the velocity, and clearing the manual_control_flag.

2. I2C Compass and GUI Feedback Costates

The program attempts to communicate with the I2C digital compass and read in the magnetic heading. A feedback costatement alternately sends the saved magnetic heading variable or robot position data to the GUI. Because the GUI expects position to be reported in latitude and longitude, the robot's Cartesian position is used to derive the reported position in the DR Costate.

3. Sonar Ranging Costate

The BL2600 does not trigger sonar ranging attempts, per se. Rather, the robot operating program uses the Sonar Ranging Costate to conduct RS232 serial communications with the Sonar Sensor Head Controller. The costate begins by flushing any data present. Then it uses Dynamic C's waitfor function to periodically check for the presence of serial data from the PIC serving as the sensor head controller microprocessor. When data is initially detected, the operating program allows the complete sonar range sentence to be transmitted and then calls the getSonarRanges function to read the data present in the serial buffer. In this manner, the CPU can simultaneously continue executing other costates while the relatively slow peripheral communications take place. The sonar ranges to objects in the twenty azimuth bins around the robot are saved to a global array. Ranges are saved in centimeters.

4. Obstacle Avoidance Costate

This costate is active only if the operator has not taken manual control of the robot. It tests the sonar range data in the global array above using the checkSonarRanges function. IR range data, in the form of floating point values of the reported analog

voltages, are also checked using the irCloseContact function. If the sonar function detects an object within the CLOSERANGE limit, equal to 40 cm, and it is within the azimuths in the robot's direction of travel, it sets the global variable closeContact, which triggers obstacle avoidance. If the IR function returns a value indicating an obstacle along the 300º azimuth, obstacle avoidance will also be triggered. Obstacle avoidance is a rudimentary random walk. The Creature slows, stops, and then rotates in place for a random period using the doRandSpin function. If the obstacle was detected with sonars, then sonar ranges are checked for objects on either side of the 300º azimuth. Similarly, if the obstacle was detected with IR sensors, the program assumes the sonar was not able to detect the original obstacle, and IR ranges are used to test the 300º azimuth for obstacles. If the 300º azimuth is clear, the robot moves away in that direction. Otherwise it remains in the obstacle avoidance loop and executes another rotation in place.

If no close contacts are detected, the Obstacle Avoidance Costate tests the sonar data for the presence of obstacles ahead of it at ranges less than the MIDRANGE value, which is set to 90 cm. If there are objects ahead, the Creature is slowed, but not stopped. The costate resets the velocity to the default autonomous speed if no sonar contacts ahead of the robot are detected inside the MIDRANGE limit.

5. IR Ranging and Wheel Tachometer Costates

The CPU samples the analog IR ranger voltages with a function call to getIRVolts within the IR Ranging Costate. The CPU conducts I2C communications with the PIC serving as the wheel tachometer to read in the observed wheel speeds using the wheel function. The function uses I2C library functions to request the value, in Hz, of each of the three wheel speeds. The observed wheel speeds are stored to global variables in radians per second. Based on the wheel speeds reported by the tachometers, the CPU calculates the direction and magnitude of the robot's velocity vector in the robot frame of reference. It also finds the amount of chassis rotation, Ω, and sets the global newDR flag to indicate the presence of new data upon which to dead reckon the robot position.

6. Dead Reckoning (DR) and NavigationCostates

The odometry data, in the form of wheel speeds, must be converted into the frame of reference of the plane that the robot is operating on. This Earth frame of reference is assumed to be a stationary plane, and the DR Costate calls the calcVeloEarthFrame to convert the robot velocity vector from the robot frame to the Earth frame using the magnetic heading to relate the two frames of reference. The function called dr (short for dead reckon) estimates the robot's Cartesian position in centimeters from the Origin, which is set to the location of the robot when the waypoint data was transmitted. It assumes straight-line motion between each update using the velocity calculated in the Earth frame. The Navigation costate is responsible for periodically recalculating the navigation solution, the command heading, required for the robot to drive from its DR position in the Earth frame to the next waypoint. It uses a flat Earth approximation and solves the arctangent to determine the direction needed to drive to the waypoint.

7. Position Formatting Costate

This costate simply calls a function, makeFakeGPSpos that operates on the Cartesian DR position to produce a latitude and longitude. It creates a string resembling a GPS NMEA sentence.

8. IMU and Heading Hold Costates

The program's IMU Costate tests for the presence of serial data on the RS232 port connected to the IMU. If the test does not fail to indicate the presence of data, the costate tests if the data indicates the start flag of an IMU data sentence. It uses the waitfor function to allow program execution to continue briefly while allowing the IMU to send its complete sentence before reading in the acceleration and rotation rate data with the getImu function. The Heading Hold Costate operates if the global holdHeading flag is set and the robot is not stopped. The costate applies proportional and derivative control feedback to maintain heading. The costate delays operation again until the receipt of new IMU data.

9. Manual Control Costate

This costate allows the operator to remotely control the robot. If manual control data packets are present, the global flag, manual_control_flag is set. This overrides autonomous operation in various costates. The costate calls the manual_control function, which parses the manual control string received from the GUI and sets the desired direction of motion and magnitude of the velocity vector. The function calls the vector function to generate the desired motion.

10. Stuck Vehicle Watchdog Costate

A low-priority costate is used at the end of the main loop to test for a condition that indicates the robot is stuck. It tests for a condition where the robot is sending the motors non-zero speed signals, i.e., it is attempting to move, but the three observed wheel speeds, from the tachometer, are nearly zero. The tachometer allows the costate to provide a simple feedback mechanism for detecting gross speed errors. If the condition is detected and it persists for a period equal to TIMELIM, equal to four seconds, the robot initiates action to attempt to free itself.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. PULSE WIDTH MODULATION (PWM) CIRCUIT

A. EXPERIMENTAL DESIGN

Experimental observations were made to determine the linearity of the PWM circuit's response to the analog speed signal by measuring the voltage supplied to each motor. The circuit was designed to produce a 100% duty cycle PWM signal at 4.1V. This voltage corresponds to the maximum analog voltage that the BL2000 single board computer can supply. To facilitate the test instrumentation the robot needed to remain stationary during the test, so the robot was placed on wooden blocks that prevented its wheels from contacting the ground. As a result all the observations of the associated motor voltages, PWM duty cycles, and shaft speeds reflect a no load condition.

A Fluke multimeter was connected to measure each of the three motor controller's output voltage. Each multimeter's ground reference was connected to the motor controller's terminal labeled M1, and the positive lead to the M2 terminal. Another multimeter was used to measure the DC analog speed signal supplied to the PWM circuit board. The DC power supply's ground reference was connected to the electronics power supply ground. The installed electronics battery aboard the robot was used to provide electrical power to the CPU, and I2C devices via the five-volt bus.

The two motor batteries were connected in series and their series voltage was applied to the motor power bus. A mulitmeter was used to record the voltage potential between the motor bus' unregulated voltage test point on the distribution/test panel and the motor ground test point on the same panel. This measurement was done with the analog speed signal set to zero at the beginning and end of the sequentially increasing test described below.

Analog input DC voltages were applied beginning at 0.0 V and were sequentially stepped up in 0.2 V increments starting at 1.0 V. The analog speed signal was connected in parallel to the three motor speed input pins on the PCB. Thus, the circuit received the same analog voltage input for all three motors, and the motors were operated simultaneously. The robot does not utilize the motor controller's brake, so the circuit on

the PCB holds this signal low, and the direction signal was allowed to remain low. The motor controller output voltage applied to each of the three motors was recorded for each analog input voltage. Extra measurements were taken near the upper limit of 4.1 V. After the data were recorded for the 4.1 V input voltage, the input voltage was reset to zero, and the motor battery voltage observed as described above.

The motor controller output voltage test described previously included added variability since it depended on the response of the LMD18200 H-Bridge chip, the resistor, and capacitors that were installed on each motor controller. The PWM waveform results from the intersection of the PWM circuit's sawtooth signal and the analog speed signal. In turn, the PWM signal is interpreted by the motor controllers, which regulate the motor voltage applied to the motors. Thus, the PWM signal was not anticipated to depend on the motor battery voltage, and this voltage was not measured in this experiment. A follow-on experiment was conducted to measure the PWM circuit's response independent of the motor controller. This experiment consisted of applying an analog speed signal to the PCB's input and observing the PWM waveforms and their duty cycles that were produced by the PWM circuit.

The PWM output signals were disconnected from the motor controllers' inputs. A Tektronix TDS3034 oscilloscope was used to measure the duty cycle of the three PWM signals. The scope probes were connected to the PWM printed circuit board (PCB) speed signal output pins and grounded to the motor battery ground test point at the distribution/test panel. The Tektronix oscilloscope's built-in percentage high duty cycle measurement was used. A multimeter was used to measure the analog input speed signal as in the above test.

As above, the analog speed input signal was connected in parallel to the three PWM PCB input pins. The same analog DC voltage was applied to the PCB's three speed inputs starting at 0.0 V. Beginning at 1.2 V the input was sequentially stepped up in 0.1 V increments to 4.1 V. For each analog input speed signal, the percent high duty cycle measured by the Tektronix oscilloscope was recorded.

B. PWM CIRCUIT EXPERIMENTAL OBSERVATIONS AND ANALYSIS

Data gathered in the experiments were interpreted using MATLAB. Since the motor batteries' voltage will decay during operation, the voltages applied to the motors were normalized to allow more meaningful interpretation. The starting and ending motor battery voltages were averaged. Figure 38 below shows the ratio of the DC motor voltage output to the average motor bus voltage as a function of analog speed signal applied to the PWM circuit.

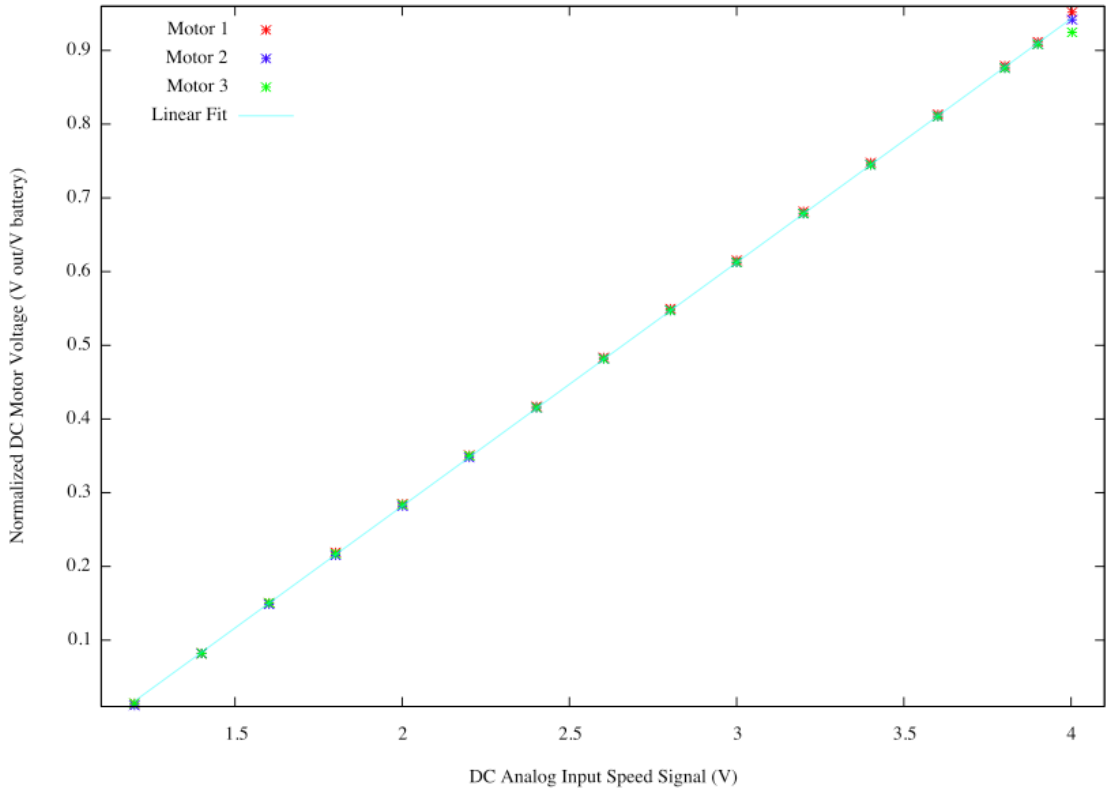

Figure 38. DC Motor Voltage output as a Function of Analog Input Voltage.

Below 1.2 V the DC motor voltage was observed to be zero. The data below 1.2 V were omitted to facilitate a linear fit. A first degree fit of the normalized DC motor voltage yielded the following relation to the analog speed signal. V_{out} is the measured output voltage, and V_{avg} battery is the average battery voltage, i.e., the maximum value attainable. V_{speed} is the independent variable, the analog speed signal applied. Of note, the motor voltages observed from the number one motor controller were consistently 0.07 to 0.06 V higher than those observed from numbers two and three.

$$\frac{V_{out}}{V_{avg.battery}} = 0.330\left(V_{speed}\right) - 0.379$$

Oscilloscope data are interpreted below. Figure 39 shows the duty cycle of the PWM waveforms as a function of the analog speed signal. The figure includes data below 1.2 V, and one can see the duty cycle goes to zero in this region.

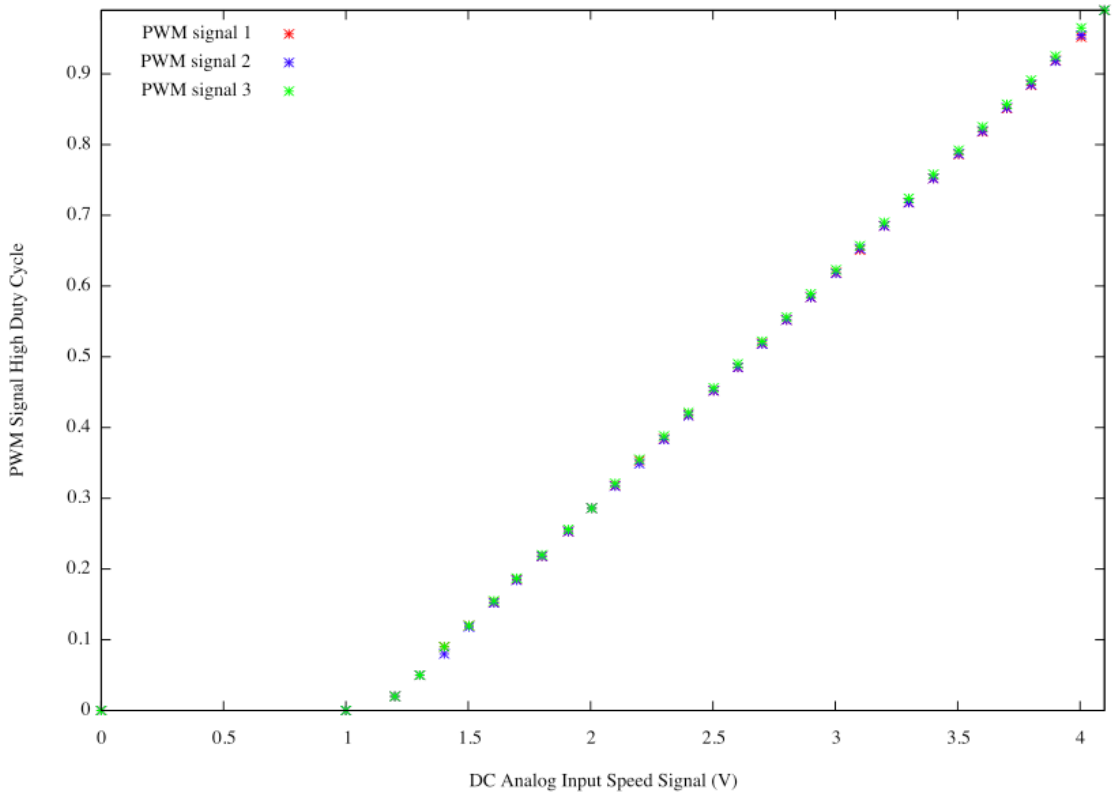

Figure 39. PWM Duty Cycle as a Function of Analog Input Voltage.

The PWM duty cycle data showed that the number three signal was consistently 0.4 to 0.6% higher than the numbers one and two signals. Figure 40 illustrates that the standard deviation of the three duty cycles were acceptably small but increased with increasing analog input voltage. Considering the observation above, there is no evidence that the PWM signals were responsible for the increased output voltage of the number one motor controller relative to the other two. Rather, the researcher suspects the number three motor controller produced the discrepancy.

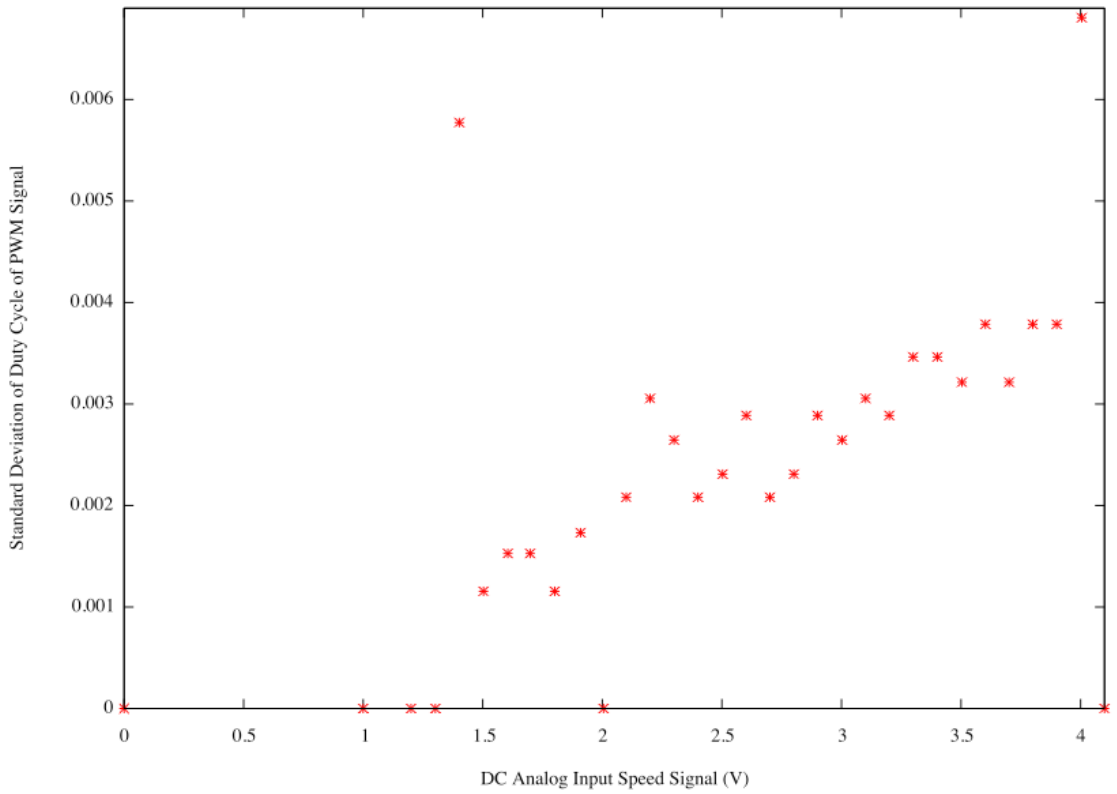

Figure 40. Duty Cycle Variation with Increasing Analog Input Voltage.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. SONAR SENSOR HEAD

Because of the robot's holonomic motion, such motion could result in a collision with an obstacle while translating in any direction. To prevent this, a sensor was developed that could provide coverage of all azimuths around the robot. Additionally, complete azimuthal coverage was necessary to implement Virtual Force Field (VFF) or Vector Field Histogram (VFH) obstacle avoidance methods used by Borenstein [42, 43]. Ultrasonic sonar was chosen because it was inexpensive, tolerant to timing errors, accurate in range, and proven over decades of robotic research.

Complete azimuthal coverage is inversely related to the beamwidth; wider beam patterns provide coverage using fewer transducers or fewer sonar ranging attempts. Wider beamwidths, however, produce imprecise azimuthal detections. Narrower beamwidths, conversely, require more sonar ranging attempts. Increasing the number of ranging attempts slows the update rate of the sensor system but provides more fine-grained azimuthal information on detected objects. The electrostatic Polaroid transducer, with its nominal 15º beamwidth, was chosen for the Creature. When paired with the Senscomp ranging module, it provides a simple digital sonar solution and offers a good compromise of directionality and coverage.

In the past, robots such as CARMEL have addressed the problem of complete azimuthal coverage with a fixed installation of 16 to 18 sonars distributed at regular angular intervals about the periphery of the robot [42]. This solution was simple and required no moving parts, but its cost, complexity of installation, weight, etc were not optimal. The Creature's rotating sonar sensor head combined the desirable aspects of the Polaroid transducer's beam pattern, but did so with four orthogonally mounted transducers. In place of 16 fixed devices, the sensor head achieves complete coverage by mechanically scanning the sonar transducers in azimuth. The sensor head consists of an RC servo motor, a square perf board mounting plate, and four transducers.

A. SONAR RANGING PATTERN

Figure 41 shows the scan pattern of the sensor head relative to the robot's X,Y reference frame. Sonar number one is oriented 60º from the robot's Y axis when at position three. Note, times reflect the position of the sonar detector head after it has reached the position but before firing the sonars. The times are based on optimized delays when operated with the final, adjusted version of the microcontroller's code.

The 360º surrounding the robot were divided evenly into twenty, 18º sectors by positioning the servo sequentially to five azimuths and ranging the four orthogonally-mounted sonar transducers once at each position. After ranging the sonars at one position the servo was moved counter clockwise and the process repeated. After the four sonars were fired at position number five, the servo moves clockwise to position number one and the cycle begins again.

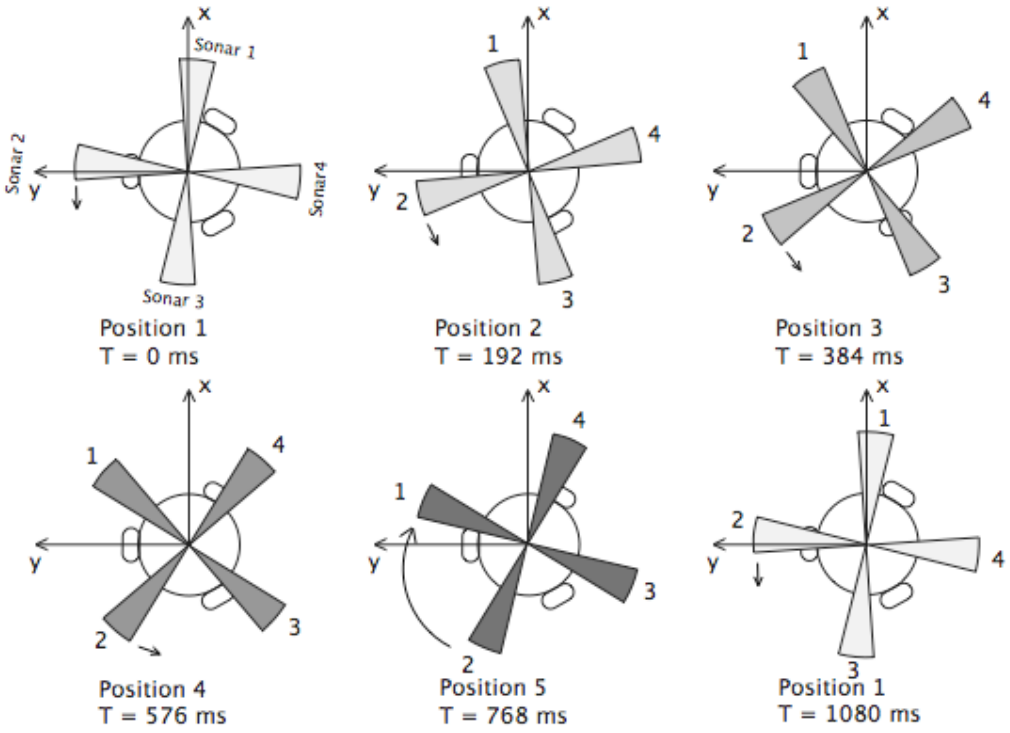

Figure 41. Sonar Sensor Head Scanning Sequence.

The RC servomotor was controlled by the sonar sensor head controller, which contains a PIC16F690 microcontroller. In all experiments, the PIC assembly code delayed 33 μs while listening for the return echo plus 20 μs to eliminate mutipath detections between each ranging attempt. The delay between sonar firings included an additional 80 μs when moving the servo motor. The expected time to complete ranging 20 sectors was 1460 μs. The sonar sensor head controller was responsible for the following actions:

- generating the servo positioning signal
- triggering the sonar firings
- finding the TOF of the pulses
- storing the TOF data
- downconverting the 16-bit TOF data to integer, one byte format
- communicating the one byte data via RS-232 to the robot's CPU

B. SONAR SENSOR HEAD EXPERIMENTAL DESIGN

A number of experiments were conducted to measure the performance of the sonar detector head. The experiments described below were designed to find the following characteristics:

- operation of the four transducers
- coefficient to convert the sensor's one byte data to centimeters
- range resolution
- minimum detectable range
- maximum detectable range
- angular resolution of the servo scanning mechanism
- cycle time required to range all sectors

The Polaroid transducer's acoustic beam pattern had been studied and well documented. No effort was made to duplicate this work; rather observations were made of the PIC microcontroller's output data and the mechanical operation of the sensor head with the servo motor operating.

All of the experiments described below were conducted in the Naval Postgraduate School's acoustic anechoic room. The room is designed for acoustic frequencies below the 49 kHz ultrasonic frequency of the Polaroid transducer. It was immediately observed that the room's walls reflected sufficient acoustic energy to register as valid echoes even in the absence of the test targets. No objects, aside from test targets, were allowed within two meters of the center of the robot. Walls were 2.2 to 3.5 m from the sonar transducers. Test targets were Ø 0.5 inch solid steel rods, roughly 2 feet long. The targets were positioned vertically. The air temperature during the experiments was 20.5 ºC.

During the experiments the robot remained stationary, and an AC to DC power supply was used to supply DC voltage to the electronics power bus in place of the electronics battery. The first-generation five-volt bus was energized to power the sonar sensor head controller. The five-volt bus' servo switch was energized when needed for experiments that required scanning the sonar transducers in azimuth. Stationary sonar sensor head observations were done by leaving the servo un-powered. The motor power bus remained off, and motors were not operated. The robot's BL2600 CPU and I2C bus power remained off.

In place of the CPU, a laptop computer with an RS-232 serial port was connected via serial cable to the serial port on the sonar sensor head controller PCB. The Hyperterminal application bundled with the laptop's Windows OS was used to observe the serial data transmissions. The researcher observed the ASCII characters transmitted from the PIC microcontroller and then translated them into deicmal integer values to interpret the TOF.

The data were one-byte values; hence one would expect valid data over the range zero to 255. The PIC, though, is programmed to send the hexadecimal value 0x02 for any received echo with integer TOF value greater than 250. This allows it to use the ASCII characters from 251 to 255 as control flags. For example, ASCII 254 and ASCII 251 indicate the start and stop, respectively, of its data word. Sonar blanking of the

transmitted pulse governs the minimum detection range, which, in turn, limits the expected minimum one-byte value. The minimum one-byte value was expected to be approximately 19.

The first observation made was a control with the sensor head in a static condition and no targets present. No objects were observed within two meters of the robot. The sonar sensor head's TOF data byte was recorded.

Next an experiment was conducted to verify the operation of the four transducers and to determine the linear coefficient needed to convert the one-byte TOF value to a range in centimeters. The test was static, the servo left unpowered. A target was placed at 100 cm distance in the center of each transducer's beam pattern, and the sonar sensor head controller's output was observed. After observing the reported TOF data byte, the distances to the four targets were verified. Two targets required small distance adjustments, as they were not precisely placed. The targets were adjusted and the TOF range bytes were recorded.

A brief test was done to determine range resolution. Following the operational test above, two targets were moved to 98 cm, while two were left in their original positions as controls. The range data bytes were recorded for the four targets.

A minimum range experiment was conducted to determine the coefficient described above and the minimum detectable range. The four targets were removed and the sensor head was reoriented manually before the minimum range test was done. With the sensor head stationary, a target was placed in the center of one of the transducer's beam patterns at 50 cm. The distance was decreased to 30 cm, 24 cm, and then 20 cm and the TOF data were recorded at each distance. At 20 cm the target's tripod legs contacted the robot's wheel and prevented placement closer than 20 cm.

A maximum range test was done in much the same manner as the minimum range test. The first experiment was attempted with a target at two meters and within about 30 cm of the corner of the room, but the proximity of the room's corner to the target prompted the researcher to abort the test and place the target in the center of the anechoic room. Beginning at 2.0 m, a target was placed in the center of the same transducer used

for the minimum range test. At each distance, the range data byte was recorded. The target was repositioned to 2.2 m, 2.3 m, and 2.4 m. At the completion of the maximum range test, the target was removed, and the range byte recorded with no target present.

Two experiments were conducted to assess the mechanical scanning of the sensor head. Unlike previous experiments, the sensor head was allowed to move to the five positions described above by powering the RC servomotor. In the first, targets were placed in the centers of three adjacent sonar ranging sectors, at 100 cm range. Measuring from the robot's Y axis, which coincides with the number one motor shaft, a target was placed at 24º, 42º, and 60º. These positions correspond to the predicted centers of three sectors, or azimuth "bins." After observing the reported range data three times, the center target at 42º was removed and the range data were recorded another three times. The test was repeated with three targets placed -30º, -12º, and 6º relative to the robot's Y axis at 100 cm.

The last azimuth scanning experiment consisted of placing one target at 1 m range and varying its azimuthal position to determine which sector would register the contact. The target was initially placed at -30º relative to the Y axis. Because the sonar's beam pattern is approximately 15º, moving the target 7.5º off the centerline of the sector was expected to produce no contact. The target was repositioned to -37º, -39º, and then to -43º relative to the Y axis while maintaining range constant at 1m. The range data were recorded for each sector in the vicinity of that expected to register the target's presence. Analysis of the sector edge experiment prompted the researcher to conduct it a second time. In the second experiment, the target was initially placed at 1 m along the -30º azimuth. It was moved counter clockwise to -39º, -43º, and -47º positions. Then it was placed at -23º and moved to -12º and 0º.

A simple experiment was conducted to verify the time required for the scanning detector head to complete a cycle of 20 sonar-ranging measurements. The time for the sensor head to return to its initial position was measured and averaged.

C. SONAR SENSOR HEAD EXPERIMENTAL OBSERVATIONS

The stationary control yielded surprising results. The acoustic anechoic room was expected to offer a pristine environment in which only echoes from the test targets would be detected by the Senscomp sonar modules. During the control, however, the sonar sensor head controller reported the one-byte decimal values 174, 173, 185, and 198. These likely corresponded to echoes detected from the room's walls.

The 1-meter static test with four targets followed. Three sonars reported 95, and one sonar reported 94. The range to the number one target, the target closest to the Y axis, was measured at 101.5 cm, and it was repositioned to 100 cm. The test was repeated with three sonars reporting 94 and one reporting 95. The range to the number four sonar target was re-measured at 100.5 cm. After repositioning this target to 100 cm, the reported ranges were 94 for all sonars. This reported range data was constant for the duration of the observation, approximately 30 s.

Targets number one and four were moved to 98 cm. The observed range data bytes were 93 for targets one and four, and 94 for targets two and three. Observed range resolution in this test was 2 cm. Expected resolution is limited by the process in the assembly code that downconverts the sonar TOF using the PIC's sixteen bit count of its TIMER1 oscillator to an eight-bit value. Operating at 8 MHz, the PIC's TIMER1 oscillator has a period of 0.5 μs. The downconverting process discards the most significant bit, 215, and the least significant bits 20 through 26. The product of 27 and 0.5 μs, the oscillator's period, yields a 64.0 μs period for the 27 bit, which one can multiply times the speed of sound in air to obtain 2.17 cm per 27 period. The product of TOF and speed of sound must be halved to give range, which yields 1.09 cm per 27 period. The least significant bit (LSB) reported by the PIC is the 27 bit, so the LSB transmitted via RS-232 in the output data represents roughly 1 cm range.

Observed range data bytes agreed with the expected results. The table below summarizes the observations from the tests above and the minimum range test. In the minimum range test, valid ranges were reported to the design limit of 20 cm, which roughly corresponds to an object in contact with the perimeter of the robot's circular chassis. The reported range data byte did not vary during any of the observation periods.

Target Range	Data byte reported by microcontroller
20 cm	21_{10}
24 cm	25_{10}
30 cm	30_{10}
50 cm	48_{10}
98 cm	93_{10}
100 cm	94_{10}

Table 4. Reported Range Byte Value vs. Measured Range.

The ratio of target range in centimeters to the one-byte range value is proportional to range. If one desired to optimize the sonar detector for precision at a particular range, one could select the ratio corresponding to that operating distance. Additionally, this coefficient will vary with the speed of sound in air; thus it is inversely proportional to the air temperature. This researcher has averaged the values from 100 cm to 50 cm to provide the operating program the needed coefficient to convert the reported data byte to centimeters.

Measuring the range to the targets in the maximum range test was less precise than above, so it is not recommended that one include the data in determining the multiplicative constant. The observed data at 2.0 m and 2.2 m were 185 and 198, respectively. Multiplying by the coefficient derived as described above gives 196 cm and 210 cm. The observed data byte with the target at 2.3 m and 2.4 m varied around 209 and 210 and was indistinguishable from the control with the target removed. It is likely the sonar transducer was detecting returns from the wire mesh floor of the room. Multiplying the nominal height of the sonar transducer, roughly 25 cm, times the tangent of the half-angle beam width, 7.5º, indicates one might expect the sonar to detect the

floor at about 1.9 m. While in the anechoic room, the robot was placed on a 2x4 wood block. The sonar transducers were roughly 3 cm higher than if the robot had been placed on its wheels atop a flat surface, and repeating the calculation predicts a possible echo from the ground at 2.1 m, which agrees loosely with the observations.

The results of the azimuthal scan tests agreed with expectations. The test, which involved placing a target in the center of three adjacent sectors, was conducted twice. Figure 42 shows the graph of observed range as a function of the sector number. The X axis is the numbered sector, which ranges from zero to nineteen. Sector zero corresponds to the eighteen-degree sector whose center is aligned with an angle six degrees from the robot's Y axis. Each sequential sector is centered eighteen degrees to the right, or in the clockwise direction as viewed from above the center of the robot. In the lower portion of Figure 42, the two graphs include sectors on both sides of the robot's Y axis, so the notation -1 has been used for the nineteenth sector and -2 for the eighteenth.

The average of the observed range byte values is plotted for the five sectors. The center three sectors indicated targets at the expected range. In the first column, one can see the three targets and the background walls in the sectors on the edges. The right column shows range as a function of sector with the center target removed. The resulting graph shows two contacts at ranges approximately 1m, but the now-empty sector shows the background range to the room's wall.

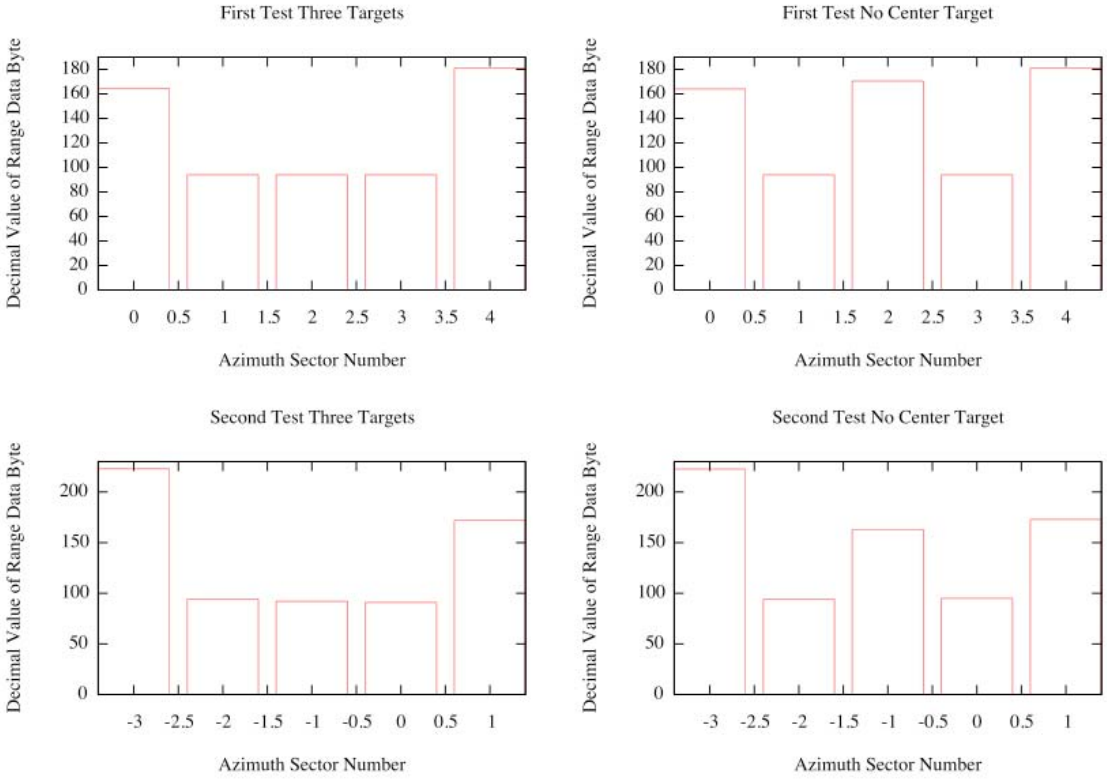

Figure 42. Detection of Targets in Three Adjacent Sectors.

Another azimuth scanning experiment was conducted to test how the sonar sensor head controller reported targets on the fringes of adjacent sectors. A single target was placed at -30º from the robot's Y axis at 1m. It was expected that this position would coincide with the center of the eighteenth sector. At -30º the target was registered by the sonar detector controller and reported in the eighteenth sector as expected. Moving counter clockwise to -37º, seven degrees off the axis of the sector, had no effect on the range data byte. At -39º, the target was also reported in the eighteenth sector. At -43º the target was reported in both the seventeenth and eighteenth sectors. Thus, although the 3dB down point on the transducer's beam pattern produces a nominal 15º beam width, the sound intensity returned from the target to the Senscomp ranging module by the transducer's side-lobes exceeds the minimum needed to register as a valid contact. The effect is an overlap between sectors for this combination of target and range.

The experiment was repeated and expanded to include attempts to find the limit of the sector on the edge closest to the Y axis. At -30º the target was reported in the eighteenth sector, as expected, and at -43º targets were reported in the seventeenth and eighteenth sectors. At -47º the target was reported in the seventeenth sector. Moving the target clockwise from -47º to -23º generated contacts in the eighteenth and nineteenth sectors. At -12º, the target was reported in the nineteenth sector only. When the target was placed along the Y axis, the reported contact corresponded to the background wall. Effectively, the target was undetected when placed at 0º.

Based on the observations of the behavior with respect to targets near the Y axis, it was concluded that the PIC's assembly code was providing an insufficient time delay, 80 μs, for the servo to move the 72º from its final position to the initial position before the sonar was fired.

The time to complete a full 360º scan was measured. The observed time, 1.7 s, agreed with the expected 1460 μs. The PIC's code was edited to remove the 20 μs delay for multipath avoidance between sonar firings. This could allow unwanted multipath detection, but the 33 μs listening period was deemed long enough to mitigate this. The servo delay between positions five and one was increased to 160 μs from 80 μs. Other servo delays were set to 60 μs. An abbreviated experiment conducted in the autonomous vehicle lab revealed a target placed on the Y axis was detectable with the modified delays.

Figure 43 shows the results of one complete scan conducted inside the Physics Department's ground floor hallway. Ranges are raw, one byte values reported and have not been converted to centimeters. The robot was stationary and placed midway between each wall with its Y axis normal to the walls. The long-axis of the hallway was oriented along the X axis. The linear features of the walls are easily visible. Note, the sonar sensor controller reports non-contact with the value 250, so one can see that no contacts were detected along the positive X axis.

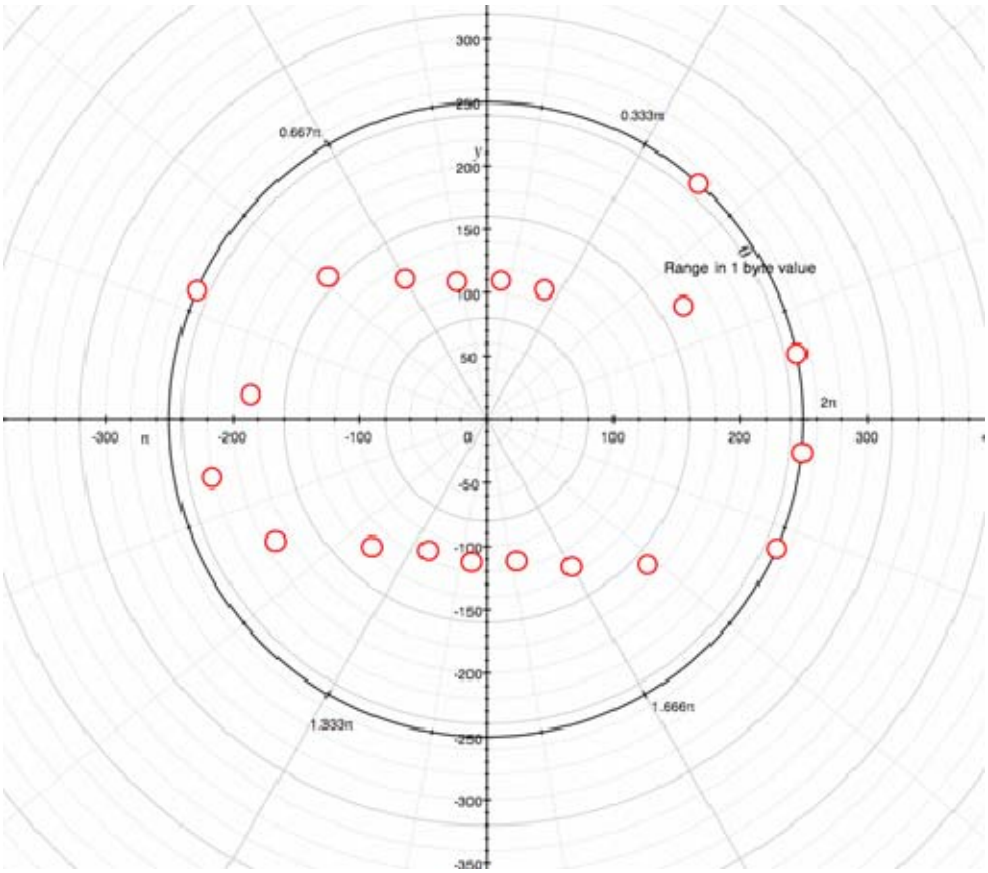

Figure 43. Sonar Azimuth Scan Showing Linear Features of Hallway.

IX. FUTURE WORK

Although the present robot's configuration represents many months of work, much work remains to be done to complete a prototype that addresses the original mission to detect FOD and remove it from the robot's operating environment. Some of the outstanding challenges represent tractable, albeit difficult, problems that exceeded the researcher's ability and knowledge as well as the time constraints associated with the project. Areas of future research can be broadly grouped into five areas: FOD detection, FOD removal, operating program improvements, GUI/human interface, and navigation.

A. FOD DETECTION

An autonomous robot represents a complex device, but developing a sensor capable of detecting small targets of a variety of materials from plastic to non-ferrous metals, in an environment with poor illumination presents future researchers with a serious challenge. A variety of methods were considered in the conceptual design phase of the Creature's development. Ultrasonic sensors were briefly considered, but it was believed their wavelengths would be too long. Ultimately, it was determined that visual wavelengths would likely be best for common FOD targets, such as washers, rivets, safety wire, etc. with characteristic dimensions on the order of a millimeter.

A conceptual sensor system was devised based on visual flow technology available with MATLAB's Simulink. An example Simulink Video and Image Processing Blockset demonstration is presently available that detects moving objects, automobiles, against a fixed background, consisting of a highway viewed from a stationary camera mounted on an overpass above the roadway [44]. In the conceptual system for the Creature, a fixed camera on the robot would stare at a swath of the surface along the direction of the robot's travel. The fixed camera's video could be linked wirelessly to an operator's laptop or workstation running the computationally intensive video image processing blockset in Simulink. The expected operating environment's surface would consist largely of steel decks covered by gray non-skid coating, so a color filter might be applied to the video stream from the fixed camera to filter out the background deck and

render objects on top of the surface more distinguishable. An optical flow Simulink block in the blockset might be applied to the filtered image to detect the objects as their apparent motion causes them to move through the camera's field of view, much like the automobiles move against the fixed background in the example above.

B. FOD REMOVAL

As a prototype, the Creature was intended to be a proof-of-concept device. The robot was intended to explore holonomic motion, navigation, and obstacle detection and avoidance. No effort was made to include a payload in the robot. Rather, it was expected that the Creature could demonstrate the feasibility of autonomous FOD detection. A follow-on robot could be built with smaller commercially-made devices, shrinking the size of many of the required components and freeing up space and weight for a payload. An engineering research area in the future might be design of a payload capable of removing FOD after it is detected. Vacuum and brush techniques might be explored as well as a method for holding the recovered FOD. Because vacuums and brush motors will almost certainly contain at least one high-current motor, efficient use of available battery power should be emphasized. Research must be done on integrating such a payload, with its high-current device, into the robot's electrical power supply busses.

C. OPERATING PROGRAM IMPROVEMENTS

There are two aspects of the robot operating program that require additional work. First, the current robot operating program is brittle with respect to peripheral device I/O. Second, the Creature has only rudimentary obstacle avoidance algorithms, and it lacks path planning.

Little provision has been made to detect if sensors are operating correctly, or if they are installed and receiving power. Portions of the Dynamic C code assume the serial communications will occur, and if they do not, the code can become indefinitely stuck awaiting the expected communication. The functions responsible for RS232 communications between the BL2600 and the sonars and between the BL2600 and the IMU need a form of watchdog time out. For example, if the expected communication with the sonar sensor head controller is not completed within roughly 60 ms, then the

function might return a value to the main function to indicate a failure. The costate responsible for communicating with the sonars should be modified to note the failure and modify its execution. If a sufficient number of consecutive failures due to lack of receipt within the allowed listening time occur, then the robot operating program might be modified to assume a sonar fault exists, stop wasting CPU cycles attempting to conduct communications that are likely to fail, and implement an IR-only obstacle avoidance behavior. Communications with I2C devices use the I2C library functions, and these have been configured in the library to limit the number of retries to prevent the CPU from spending excessively large numbers of processor cycles attempting to communicate with devices that are not responding. Regardless, a diagnostic costate might be added to check for the proper functioning of all peripheral devices on the I2C bus as well as proper operation of devices communicating via RS232.

Obstacle avoidance and path planning should be improved. The researcher was only able to develop a random walk behavior in the time available. Research was done into potential field algorithms, particularly Borenstein's virtual force field (VFF) and vector field histogram (VFH), for implementation into the robot operating program as an underlying obstacle avoidance algorithm to combine ultrasonic sonar and IR sensor range data. Borenstein's work using ultrasonic sensors identical to those on the Creature allowed motion at up to 0.78 m/s while successfully avoiding obstacles [42, 43]. A large portion of the work to implement VFF was done, but not completed or tested. Dynamic C code was written to map the sonar range and azimuth data to a Cartesian frame of reference with its origin at the center of the robot, and functions were written to update the Cartesian representation of the sonar data for straight-line relative motion between the objects and the robot. No code exists yet to handle rotational transformations required when the robot rotates with respect to the Earth frame of reference. Also, a function would need to be written to sum the repulsive forces and target's attractive force to produce the resultant force on the robot.

Without adequate orientation data about the robot's pose, the existing code cannot accurately determine the position of the robot from wheel odometry. This problem relates to the separate navigation problem below. Thus, without accurate orientation, navigation accuracy is poor, and as a result, path planning is moot. The researcher believed it was pointless to attempt to navigate from an erroneous position to a waypoint position, so path planning was stricken from the robot operating program. A random walk was implementated, instead. If navigation can be improved in the future, path planning should be re-introduced into the operating program code to extend the Creature's ability to conduct other tasks. Future research could simulate the motion of an omnidirectional robot using a random walk to search an area. Simulations could compare area covered by random motion to area covered using a planned search. Future research could determine if random searches for FOD are sufficient to solve the problem of detection and removal. If so, cost savings could be realized, and a simplier follow-on robot constructed.

D. GUI/HUMAN INTERFACE

The Java language GUI provides a useful means to monitor and control the Creature, but its background as a front-end for control of robots in an outdoor environment has hobbled it when the researcher modified it for use indoors with the Creature. Fundamentally, the GUI expects the robot to use GPS for absolute positioning. Clearly this is a bad assumption when dealing with robots operating indoors, inside the interior of a ship, underground, or in any other environment where GPS is unavailable. A work-around was to take the Creature's dead reckoned position, which was stored as a Cartesian position from the origin, a known starting point in the Earth frame, and periodically convert the orthogonal East and North distances from the known point into a latitude and longitude. The GUI could be modified in the future to accept either latitude/longitude or Cartesian position information.

Manual control of the Creature with the existing GUI is adequate, but slow compared to the robot's capabilities. Specifically, the ability of the operator to interpret the robot's pose and send the correct driving commands limits the speed at which the

robot can move through its environment. As a holonomic platform, the Creature has no front or rear. The lack of such arbitrary directions means an operator controlling the Creature has many choices when trying to move the robot. Additionally, the robot provides poor visual cues as to its orientation when viewed at distances exceeding three to four meters. A valuable area of future work would be the creation of an improved user interface, complete with joystick controller, to vector the robot and a jog-shuttle to command robot chassis rotation. Additionally, future researchers might implement an improved display, possibly a three-dimensional one fixed to the Earth reference frame, to allow the operator to easily interpret the robot's pose and quickly apply the desired driving commands.

E. NAVIGATION

The problem of determining robot position without GPS was difficult to solve. Previous SMART program robots have assumed GPS would be available and depended on it for the primary source of position information. In an environment without GPS, researchers are left with traditional methods such as dead reckoning, e.g., from wheel odometry, and beacon methods. A vibrant and exciting area of research could be the implementation of a newer navigation method based on robotic vision. The researcher believes the computational load to derive robot motion from a video stream would vastly exceed the limits of the BL2600's processor, so a remote camera method such as the one demonstrated by Shimada et al. might be employed [45]. In this position correction method, a fixed remote overhead camera observes a scene in which the robot operates. From the video image's pixel data, a remote computer running a vision algorithm determines the robot's position and orientation. The position is communicated to the robot periodically. Between updates, the robot uses dead reckoning techniques to maintain its position.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A – PULSE WIDTH MODULATOR AND OPTICALISOLATION CIRCUIT

Figures 44 and 45 show the schematic of the PWM and optical isolation circuit that was constructed to send analog speed signals from the BL2600 SBC to the motor controllers. The Jumper J1 was included to allow the user to choose one half of the 12.0 V power via the voltage divider to set the thresholds of the Schmitt trigger. Alternatively, one could use an external precision voltage reference connected to VCC if the user desired greater accuracy.

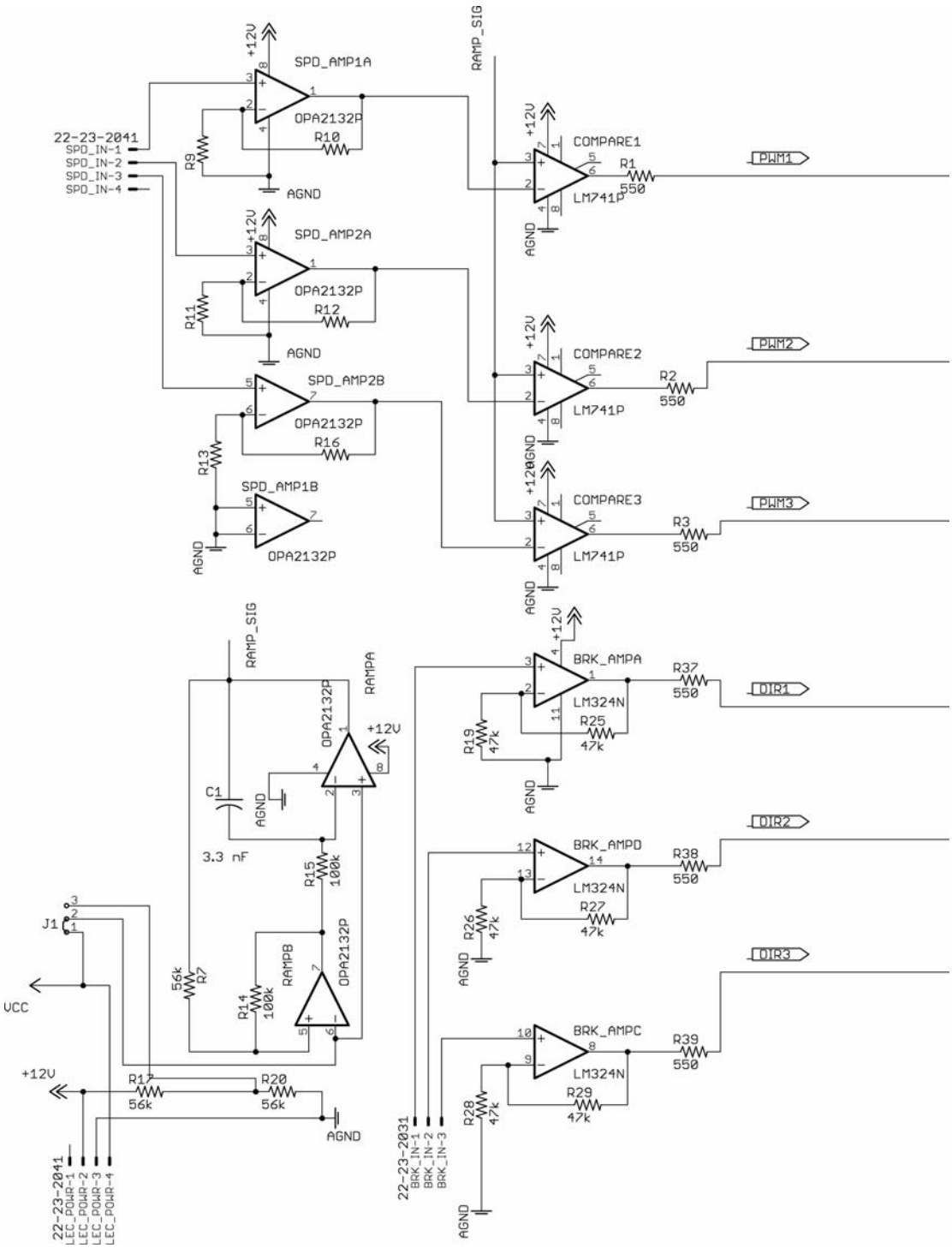

Figure 44. PWM Schematic Page 1.

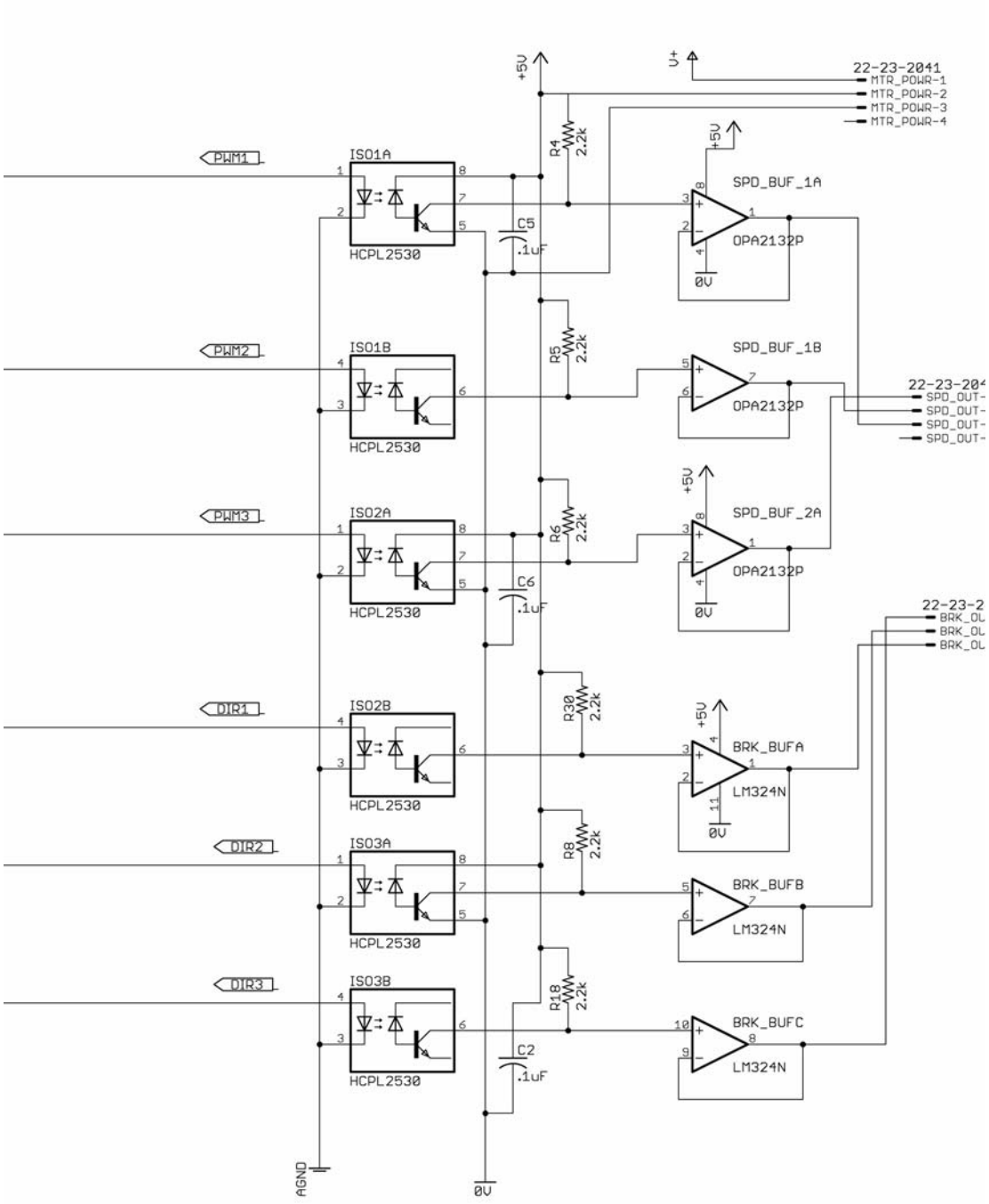

Figure 45. PWM Schematic Page 2.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B – ELECTRICAL WIRING COLOR CODES AND LABELS

A wiring color code was implemented using the code described in the table below. Orange Alfa wire, in 16 AWG, was used for main power supply wiring. It was not available in multiple colors, so extensive efforts were made to label wires for ease of maintenance. The robot's electrical wires were numbered to assist researchers in maintaining and troubleshooting electrical connections. The numbering scheme was implemented early in the development of the Creature, before many of the devices were conceived, let alone installed. Because of this, the numbering is not sequential. The numbering scheme left many numbers unused allowing them to be assigned to devices as they were installed. Some effort was made to logically group numbers by function. Five-volt electronics bus supplies were assigned alphabetical labels as a mnemonic to aid in identifying their function.

Wire Function	Color
Ground	Black
5 V supply	Red
12 V supply	Yellow
Unregulated Nominal 16V supply	Yellow
Digital Communication (example I2C)	Purple
Digital Signal	Blue
Analog Signal (example motor speed)	White

Table 5. Wire Color Codes.

Label	Function
1	IR #1 analog range voltage
2	IR #2 analog range voltage
3	IR #3 analog range voltage
4	IR #4 analog range voltage
5	IR #5 analog range voltage
6	IR #6 analog range voltage
23	Motor 1 +24 V
24	Motor 1 0V
30	Motor 2 0V
31	Motor 2 +24V
40	I2C SDA
41	I2C SCL
50	Motor 3 0V
51	Motor 3 +24V
58	Motor Bus Ground
59	Motor Bus 24V Unregulated
60	Motor Bus Ground
61	Motor Bus 5V Regulated
62	Unregulated Input to 7812
63	7812 12V Regulator Output
64	Electronics Bus Ground
65	Electronics Bus 12V Regulated
66	Unregulated 16V Electronics
B	RabbitLink Supply 16 V Unreg.
C	I2C Compass 5.0 V Regulated
D	BL2600 +K external pullup voltage
I	IMU 5.0 V Regulated
R	Router 5.1 V Regulated
S	Sonar 5.1 V Regulated
V	Sonar Servo 5.1V Regulated
W	I2C Wheel Tach 5.0 V Regulated
X	IR Sensors 5.0 V Regulated

Table 6. Electrical Wiring Function to Label Cross-reference.

APPENDIX C – CREATURE OPERATING MANUAL

The complex and unique nature of the Creature's electrical power supply prompted the researcher to write a short user's manual for the robot. The purpose of the manual is to allow future researchers to continue further research into operating program code enhancements, navigation, and obstacle avoidance without having to spend time tracing wires and deciphering the functions of various switches and fuses. Familiarity with the java GUI, pioneered by Uzun and running on the operator's laptop, is assumed.

Operating the Creature involves two start-up sequences: the vehicle and the laptop GUI. Before starting up, though, please ensure the Creature's batteries have been charged. See the Motor Battery Charging and Electronics Battery Charging Procedures for details. Following the start up sequence given in Figures 46 through 49 will help ensure sensitive electronic devices are not subjected to voltage transients.

MOTOR BATTERY CHARGING

Main Power Panel Motor Switch.....OFF
Motor Battery Charge Panel Charge Switch.....OFF/CHARGE
Motor Battery Charge Panel Series Switch .....OFF
Battery A Positive to Charger Positive Lead.....CONNECTED
Battery A Negative to Charger Neg. Lead.....CONNECTED
Repeat for Battery B

Note

Creature uses 10 C-cell NiMH batteries. Capacity 4000 mAH. Set Charger appropriately.

Battery A Charger.....ON
Battery B Charger.....ON

Note

Charging time varies depending on initial charge state. Charging times of 3 to 4 hours are common at 0.8 A charging current.

When charging complete...
Battery A Charger.....OFF
Battery B Charger.....OFF
Battery A Positive Lead.....DISCONNECT
Battery A Neg. Lead.....DISCONNECT
Repeat for Battery B
If robot is to be operated...
Robot Pre-Start Sequence.....COMPLETE

ELECTRONICS BATTERY CHARGING PROCEDURE

Note

Electronics Battery has an isolated supply and ground from the Motor Battery. All batteries may be charged simultaneously.

Caution

Failure to turn off MPP Motor Power Switch prior to shutting down CPU can result in uncommanded motor operation at full speed. Damage to robot could result if not on blocks.

Main Power Panel Motor Switch.....OFF
Main Power Panel CPU Switch.....OFF
5V Bus Switches.....OFF
Main Power Panel BUS Switch.....OFF
MPP Electronics Battery Supply Cord.....DISCONNECT
Electronics Battery Supply Cord at Batt.....DISCONNECT & REMOVE

Figure 46. Creature Operating Manual Page 1.

PowerStation100 Charger..... PLUG INTO BATTERY

PowerStation100 Charger..... PLUG INTO AC OUTLET

Note

Electronics Battery charge times can be as long as 4 hours. Green 100% LED indicates full charge. See PowerStation100 manual for more information.

Electronics Battery Visual Indicator..... MONITOR FOR RED LED

When Charging completed....
PowerStation100 Charger..... DISCONNECT FROM AC
PowerStation100 Charger..... UNPLUG FROM BATT.

Electronics Battery Supply Cord at Batt..... INSTALL INTO BATTERY

Note

Leave Electronics Battery Supply Cord disconnected from MPP until ready to operate. Connecting to the MPP can allow linear voltage regulators to begin drawing current and drain battery when robot is not in use.

ROBOT PRE-START

Motor Battery Charging..... AS REQUIRED
Electronics Battery Charging ..... AS REQUIRED

Figure 47. Creature Operating Manual Page 2.

ROBOT START

Main Power Panel Motor Switch.....OFF

Caution

Applying 24V to motors without CPU operating can result in uncommanded motor operation at full speed. Damage to robot could result if not on blocks.

Motor Battery Charge Panel Charge Switch.....ON/RUN
Motor Battery Charge Panel Series Switch .....SERIES

Aux. Power Cord at MPP.....DISCONNECT AS REQ.
Wires, leads, programming cables, etc.....CHECK REMOVED

Note

If robot is to be operated motionless or on blocks, test leads and programming cables can be left installed.

Main Power Panel BUS Switch.....OFF
Main Power Panel CPU Switch.....OFF
Electronics Battery Supply Cord at Batt.....PLUGGED INTO BATT.
Electronics Battery Supply Cord at MPP .....INSTALL
Main Power Panel BUS Switch.....ON
5V Bus Sonar Switch.....ON
5V Bus Servo Switch.....ON

Note

Failure to apply power to the Sonar Sensor Head before activating the Router will cause the Router to hang or fault. If this happens, cycle router power. If CPU is operating, it may be necessary to cycle power to CPU also.

5V Bus Router Switch .....ON
Laptop WiFi and TCP/IP Setup.....COMPLETE

Note

If the Creature wireless A.P. shuts down due to insufficient 5V supply, the Creature WiFi LAN will become unavailable on the laptop. Investigate 5V Bus supply.

5V Bus RabbitLink Switch.....ON
Main Power Panel CPU Switch.....ON
5V Bus I2C Compass/Tach Switch.....ON
5V Bus IMU/IR Switch.....ON
Main Power Panel Motor Switch.....ON

Figure 48. Creature Operating Manual Page 3.

ROBOT SHUTDOWN

Main Power Panel Motor Switch.....OFF
5V Bus IMU/IR Switch.....OFF
5V Bus Sonar Switch.....OFF
5V Bus Servo Switch.....OFF
5V Bus I2C Compass/Tach Switch.....OFF
5V Bus RabbitLink Switch.....OFF
5V Bus Router Switch.....OFF
Main Power Panel BUS Switch.....OFF
Main Power Panel CPU Switch.....OFF

Motor Battery Charge Panel Charge Switch.....OFF/CHARGE
Motor Battery Charge Panel Series Switch .....OFF

LAPTOP WIFI AND TCP/IP SETUP

Laptop turn on.....AS REQ.
Mac Airport or Windows Wireless Networking.....ON
When Creature WiFi A.P available....
Creature Access Point (A.P).....CONNECT TO
DHCP.....OFF
IPv4.....MANUAL
TCP/IP Router IP Address.....192.168.4.47 SET
Laptop IP Address.....192.168.4.37 SET

Figure 49. Creature Operating Manual Page 4.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D – WHEEL TACHOMETER ASSEMBLY LANGUAGE CODE

The PIC assembly language code below, specifically the ISR and I2C communications code and subroutines, has been used with the permission of the code's original author to implement the tachometer functionality of the device installed in the Creature [38].

```
; pic3Wheel20NOV.asm
; By Kirk Volland
; 11/20/2007
; based on I2C ISR code written by
; by Michael Gasperi
; used with the author's permission
; 6/11/2007  fixed per 16f690 errata on I2C port

; times for 250 ms and checks for changes (transitions hi to lo or lo to hi)
; on three digital input pins.  Counts transistions, multiplies by 2.
; When receives i2c read it writes the count to i2c bus.

#include <p16f690.inc>     ; Change to device that you are using.
#define       NODE_ADDR   0xD0   ; I2C address of this node 64 decimal

#define WHL1OLD      wheelTachLast,5
#define WHL2OLD      wheelTachLast,4
#define WHL3OLD      wheelTachLastB,7

#define WHL1  wheelTachNow,5
#define WHL2  wheelTachNow,4
#define WHL3  wheelTachNowB,7

;*****************************************************************
; Variable declarations
;*****************************************************************
       cblock 0x20
WREGsave
STATUSsave
FSRsave
PCLATHsave
Temp        ; used to decode ISR
RegSel              ; RegSelect
DataRegs:7
invalidWheelDat
startValues
wheelTachNow
wheelTachNowB
wheelTachLastB
wheelTachLast
wheel_1_cnt
wheel_2_cnt
wheel_3_cnt
i
       endc
```

```
       org 0
       goto   Startup      ; 0x0001
       org 004
       goto   ISR          ; 0x0004

;--------------------------------------------------------------------
; Main Code
;--------------------------------------------------------------------

Startup
       ; set up oscillator freq
       bsf          STATUS,RP0          ; Register Bank 1
       bcf          STATUS,RP1          ; Register Bank 1
       movlw        b'01110001'         ; 8 MHz internal oscillator
       movwf        OSCCON

       bcf          STATUS,RP0
       bcf          STATUS,RP1
       clrf         PORTC
       banksel      TRISC
       movlw        0xFF          ; C all input
       movwf        TRISC          ; config port C
       movlw        0xFF          ; all input
       movwf        TRISA         ; config port A
       movlw        0x00          ; A2D off
       banksel      ADCON1
       movwf        ADCON1
       movlw        0x00          ; all port a digital
       banksel      ANSEL
       clrf         ANSEL
       movlw        0x00          ; select channel 1 on A2D
       banksel      ADCON0        ; A2D off
       movwf        ADCON0

       banksel      PCON          ; power control
       bsf          PCON,NOT_POR
       bsf          PCON,NOT_BOR
       banksel      PORTB
       clrf         PORTB         ; Clear port B
       banksel      PIR1
       clrf         PIR1           ; Clear Interupts
       banksel      TRISB
       movlw        0xff
       movwf        TRISB         ; Port B all input
       banksel      ANSEL
       clrf         ANSELH        ; Port B all digital
                                  ; set up TIMER1
       bcf          STATUS,RP0
       bcf          STATUS,RP1    ; Bank 0
       clrf         T1CON         ; reset it
       clrf         TMR1H         ; clear the 2 bytes of counters
       clrf         TMR1L
       movlw        b'00110000'   ; prescaler 1:8 for long timing 250 ms
       movwf        T1CON         ; Use internal clock source. Keep TIMER1 off


       bcf          T1CON,0       ; TIMER1 disable
       clrf         PIR1          ; clear TMR1 overflow flag
       clrf         TMR1H         ; and Timer1 hi and lo count bytes
       clrf         TMR1L
       movlw        0x0B
```

```
        movwf         TMR1H         ; load TIMR1 with value = 3036
        movlw         0xDC          ; so count up from 3036 to 65535
        movwf         TMR1L
        bsf           T1CON,0       ; enable TIMER1 and start counting up

        banksel       SSPCON
        movlw         0x39          ; setup ssp module workaround sequence
        movwf         SSPCON        ;
        movlw         0x36          ; Setup SSP module for 7-bit
        movwf         SSPCON        ; address, slave mode
        movlw         NODE_ADDR     ; set Node Address
        banksel       SSPADD
        movwf         SSPADD
        banksel       SSPSTAT
        clrf          SSPSTAT       ; clear SSP status
        banksel       PIE1          ; Enable interrupts
        bsf           PIE1,SSPIE
        bsf           PIE1,TMR1IE   ; Enable timer1 interrupt
        bsf           INTCON,PEIE   ; Enable all peripheral interrupts
        bsf           INTCON,GIE    ; Enable global interrupts


Do250msLoop

        bcf    STATUS,RP0
        bcf    STATUS,RP1
        clrf   wheelTachNow  ; clear the variable for holding PORTC results
        clrf   wheelTachNowB
        movf   PORTC,w       ; put the current tachsignal from POTRC into
                             ; WREG
        movwf  wheelTachNow  ; make a copy of current values in case it
                             ; changes in next few cycles
        movf   PORTB,w
        movwf  wheelTachNowB
Test1
        btfsc  WHL1
        goto   WhlTach1Set   ; do this if current wheel 1 tach signal is LO
        btfss  WHL1OLD       ; was the last wheel 1 tach signal LO?
        goto   Test2         ; if both are LO, then no change, go to wheel 2 test
        incf   wheel_1_cnt,f ; increment if new == 0 and old == 1
        goto   Test2
WhlTach1Set
        btfsc  WHL1OLD       ; current tach is HI.  Was last tach HI, too?
        goto   Test2         ; if current == 1 and old == 1, then no change, go
                             ; to Test3
        incf   wheel_1_cnt,f       ; new not equal to old, so increment the count
Test2
        btfsc  WHL2
        goto   WhlTach2Set   ; do this if current wheel 1 tach signal is LO
        btfss  WHL2OLD       ; was the last wheel 1 tach signal LO?
        goto   Test3         ; if both are LO, then no change, go to wheel 3 test
        incf   wheel_2_cnt,f       ; increment if new == 0 and old == 1
        goto   Test3
WhlTach2Set
        btfsc  WHL2OLD       ; current tach is HI.  Was last tach HI, too?
        goto   Test3         ; if current == 1 and old == 1, then no change, go to
                             ; end. copy values to old
        incf   wheel_2_cnt,f       ; new not equal to old, so increment the count
Test3
        btfsc  WHL3
        goto   WhlTach3Set   ; do this if current wheel 1 tach signal is LO
        btfss  WHL3OLD       ; was the last wheel 1 tach signal LO?
```

```
       goto   CopyVals     ; if both are LO, then no change, go to copy values
       incf   wheel_3_cnt,f       ; increment if new == 0 and old == 1
       goto   CopyVals
WhlTach3Set
       btfsc  WHL3OLD      ; current tach is HI.  Was last tach HI, too?
       goto   CopyVals     ; if current == 1 and old == 1, then no change, go
                           ; to end of loop, copy values
       incf   wheel_3_cnt,f       ; new not equal to old, so increment the count

CopyVals
       movf   wheelTachNow, w     ; end of loop
       movwf  wheelTachLast       ; save current values as old values before
                                  ; restart loop
       movf   wheelTachNowB,w
       movwf  wheelTachLastB
       movlw  0x0B   ; count down from 27 to zero
       movwf  i      ; delay the check loop for about 40 usec before checking
       decfsz i,f    ; again to allow for signal rise times.
       goto   $-1

       clrwdt
       goto   Do250msLoop    ; Do it again


;-----------------------------------------------------------------------
; Generic Interrupt Service Routine
;-----------------------------------------------------------------------
ISR
       movwf        WREGsave     ; Save WREG
       movf         STATUS,W     ; Get STATUS register
       banksel      STATUSsave   ; Switch banks, if needed.
       movwf        STATUSsave   ; Save the STATUS register
       movf         PCLATH,W     ;
       movwf        PCLATHsave   ; Save PCLATH
       movf         FSR,W        ;
       movwf        FSRsave      ; Save FSR
       banksel      PIR1
       btfsc        PIR1,SSPIF   ; Is this a SSP interrupt?
       goto         HandleSSP
       btfss        PIR1,TMR1IF  ; Is it a Timer 1 overflow?
       goto         $            ; No, just trap here.
       bcf          PIR1,TMR1IF  ; clear interrupt for 250ms loop in timer1
       movlw        0x0B
       movwf        TMR1H        ; reload TIMR1 with value = 3036
       movlw        0xDC         ; so count up from 3036 to 65535
       movwf        TMR1L
       bcf          STATUS,C     ; since timing for 1/4 second and counting
                                 ; every half cycle
       rlf          wheel_1_cnt,w; multiply wheel count by 4 and divide by to
                                 ; 2 to get number cycles /1 sec
       movwf        DataRegs+1
       clrf         wheel_1_cnt  ; zero the current count for wheel 1
       bcf          STATUS,C     ; since timing for 1/4 second and counting
                                 ; every half cycle
       rlf          wheel_2_cnt,w; multiply wheel count by 4 and divide by 2
                                 ; to get number cycles /1 sec

       movwf        DataRegs+2
       clrf         wheel_2_cnt  ; zero the current count for wheel 2
       bcf          STATUS,C     ; since timing for 1/4 second and counting
                                 ; half cycle every
```

```
       rlf          wheel_3_cnt,w ; multiply wheel count by 4 and divide by 2
                                  ; to get number cycles /1 sec

       movwf        DataRegs+3
       clrf         wheel_3_cnt         ; zero the current count for wheel 3
       movf         PORTC,w
       movwf        wheelTachLast       ; seed the old value
       movf         PORTB,w
       movwf        wheelTachLastB      ; seed the old value
       goto         RestoreFromISR
HandleSSP
       bcf          PIR1,SSPIF
       call         SSP_Handler         ; Yes, service SSP interrupt.
RestoreFromISR
       banksel       FSRsave
       movf         FSRsave,W
       movwf        FSR                 ; Restore FSR
       movf         PCLATHsave,W
       movwf        PCLATH              ; Restore PCLATH
       movf         STATUSsave,W
       movwf        STATUS              ; Restore STATUS
       swapf        WREGsave,F
       swapf        WREGsave,W          ; Restore WREG
       retfie                           ; Return from interrupt.

SSP_Handler
       banksel      SSPSTAT
       movf         SSPSTAT,W    ; Get the value of SSPSTAT
       andlw        b'00101101'  ; Mask out unimportant bits in SSPSTAT.
       banksel      Temp         ; Put masked value in Temp
       movwf  Temp               ; for comparision checking.

State1:                         ; Write operation, last byte was an
       movlw        b'00001001'  ; address, buffer is full.
       xorwf        Temp,W       ;
       btfss        STATUS,Z     ; Are we in State1?
       goto         State2       ; No, check for next state.....
       banksel      SSPBUF
       movf         SSPBUF,W     ; Get the node addr and throw it away
       movlw        0x00         ; initalize register select to 0
                                 ; so first write will go into itself
       movwf   RegSel
       return

State2:                         ; Write operation, last byte was data,
       movlw        b'00101001'  ; buffer is full.
       xorwf        Temp,W
       btfss        STATUS,Z     ; Are we in State2?
       goto         State3       ; No, check for next state.....
       banksel      SSPBUF
       movf         SSPBUF,W     ; Get the byte into W
       banksel      DataRegs
       movwf        DataRegs     ; save the byte to 0th array location

       return

State3:                         ; First Read operation, last byte was an
       movlw        b'00001100'  ; address, buffer is empty.
       xorwf        Temp,W
       btfss        STATUS,Z     ; Are we in State3?
       goto         State4       ; No, check for next state.....
       bcf          STATUS,Z
```

```
                                ; case switch is the value of DataRegs == 3
       movf         DataRegs,w
       xorlw        3
       btfss        STATUS,Z
       goto         NotEq3
                                 ; DataRegs = 3 so send out 4th array element
       movf         DataRegs+3,w        ; save to WREG for output
       goto         State3Send
NotEq3                          ; if DataRegs != 3 is it 2?
       xorlw        2^3
       btfss        STATUS, Z
       goto         NotEq2
                                 ; DataRegs = 2 so send out 3th array element
       movf         DataRegs+2,w       ; save to WREG for output
       goto         State3Send
NotEq2                          ; default if not == 2 or 3, then DataRegs == 1
       movf         DataRegs+1,w       ; save array element 2 to WREG

State3Send
       call         WriteI2C     ; Write the byte to SSPBUF
       return

State4:                         ; Other reads operation, last byte was data,
       movlw        b'00101100'  ; buffer is empty.
       xorwf        Temp,W
       btfss        STATUS,Z     ; Are we in State4?
       goto         State5       ; No, check for next state....
       banksel       RegSel
       movlw        0xA
       call         WriteI2C     ; Write the byte to SSPBUF
       return

State5:
       movlw        b'00101000'  ; A NACK was received when transmitting
       xorwf        Temp,W       ; data back from the master.  Slave logic
       btfss        STATUS,Z     ; is reset in this case.  R_W = 0, D_A = 1
       goto         I2CErr       ; and BF = 0
       return              ; If we aren't in State5, then something is wrong.

I2CErr
       nop
       goto   $                 ; let watch dog catch this
       return

;-----------------------------------------------------------------------
; WriteI2C
;-----------------------------------------------------------------------
WriteI2C
       banksel      SSPSTAT
       btfsc        SSPSTAT,BF   ; Is the buffer full?
       goto         WriteI2C     ; Yes, keep waiting.
       banksel      SSPCON       ; No, continue.
DoI2CWrite
       bcf          SSPCON,WCOL  ; Clear the WCOL flag.
       movwf        SSPBUF       ; Write the byte in WREG
       btfsc        SSPCON,WCOL  ; Was there a write collision?
       goto         DoI2CWrite
       bsf          SSPCON,CKP   ; Release the clock.
       return
       end          ; End of file
```

APPENDIX E – SONAR SENSOR HEAD CONTROLLER ASSEMBLY LANGUAGE CODE

```
;*****************************************************************
; sonarServoHead11.asm
;
; version 11 from sonarServoHead10.asm
; Change History:
; version 11 adjusts time delay between sonar numbers 1 and 4
; to start motors moving then do delay for false echoes.
; ver7b added longer delay when moving to position 5 to give servo
; Time to move the longer distance.  Delays 160 ms.
; Changed delay when moving between sectors to 80 ms from 60 ms.
; ver7a
; Coverted back to one byte data format per range reading.
; Added one byte at end of range sentence to indicate which element in
; array is the newest (i.e. time late = 0). Other time lates can be
; figured from that one.
; Added test of 16 bit count. If MSB is set, then range > max. reportable.
; Also, if downconverted byte > 250, then sets value to 0x02 for no contact.
; Fine tuned the sonar firings, waits 40 ms between each sonar firing
; at a given position.
; Changed to 4 sonars and 5 servo positions.
; Moves sonar head to 5 positions.
; Generates 20 sonar range values
;
;
; Kirk Volland
; 19 OCT 2007
;*****************************************************************
; Hardware: PIC16F690 running at 8MHz
; Sonar 1 INIT output on RA2
; Sonar 2 INIT output on RB6
; Sonar 3 INIT output on RA5
; Sonar 4 INIT output on RB4
; Blanking Inhibit BINH output on RA4
; Combined OR'ed ECHO line input on RC5
; Servo control output on RC0
;
;*****************************************************************
; TIMER0 prescaler set to 1:16 so one 'tick' is 8 usec
; use TIMER0 for the servo pulse widths
; TIMER1 prescaler set 1:1 so one 'tick' is .5 usec
;*****************************************************************
#define ECHOPIN PORTC,5
#define PULSEONDELAY 0xB2


#include <p16F690.inc>
    __config (_INTRC_OSC_NOCLKOUT & _WDT_OFF & _PWRTE_OFF &
       _MCLRE_OFF & _CP_OFF & _BOR_OFF & _IESO_OFF & _FCMEN_OFF)

; variables
       cblock 0x20
i
j
pulseCount
SonarRange:2
SonarRanges:20
```

```
PulseCount
Position
SonarNum
tempVal
ArrIndex
       endc
; macros
servoHiMac          macro
       movlw        0x01
       movwf        PORTC
       endm

servoLoMac          macro
       nop
       nop
       nop
       nop
       movlw        0x0
       movf         PORTC,f
       nop
       nop
       nop
       endm

PingSonar4Mac macro
       movlw        b'00010000'
       movwf        PORTB
       endm
PingSonar3Mac macro
       movlw        b'00100000'
       movwf        PORTA
       endm
PingSonar2Mac macro
       movlw        b'01000000'
       movwf        PORTB
       endm
PingSonar1Mac macro
       movlw        b'00000100'
       movwf        PORTA
       endm

       org 0
       nop

;**************************  Initialization  *************************
Init
       ; set up oscillator freq
       bsf          STATUS,RP0                ; Register Bank 1
       bcf          STATUS,RP1                ; Register Bank 1
       movlw  b'01110001'                     ; 8 MHz internal oscillator
       movwf  OSCCON

       bcf          STATUS,RP0
       bcf          STATUS,RP1          ; Bank 0
       clrf   CCP1CON                   ; clear capture compare module,
                                        ; turn it off
       clrf   PORTA               ; init portA
       clrf   PORTB               ; init portB
       clrf   PORTC               ; init PORTC
       ; make all input and outputs digital
       bsf    STATUS,RP0                ; Register Page 2
       bsf    STATUS,RP1                ; Register Page 2
```

```
       clrf   ANSEL              ; all digital IO
       clrf   ANSELH
       ; set up TIMER1
       bcf    STATUS,RP0
       bcf    STATUS,RP1    ; Bank 0
       clrf   T1CON         ; reset it
       clrf   TMR1H         ; clear the 2 bytes of counters
       clrf   TMR1L
       movlw        0x0     ; prescaler 1:1 for Sonar Range timer
       movwf        T1CON   ; Use internal clock source. Keep TIMER1 off
       ; set up TIMER0
       bsf          STATUS,RP0    ; Bank 1
       movlw        b'00000011'   ; Sourced from the Processor clock
       movwf        OPTION_REG    ; Bits 2,1,0 are 011, so prescaler 1:16 or
                                  ; 8 usec per bit for Servo Pulse HI time

       ; set up pins for Inputs and Outputs
       movlw        b'00100000'   ; All output except capture comparator ccp1
       movwf        TRISC         ; RC5 input (combined ECHO all others out
       clrf         TRISB         ; port B all output
       clrf         TRISA         ; port A all output

                           ; intialize baud rate set up 57.14 kbit
                           ; Fosc = 8MHz
                           ; SYNC = 0, BRGH= 1, BRG16 = 1, SPBRG = .34
        movlw       .34
        movwf SPBRG
        bsf   BAUDCTL, BRG16            ; set BRG16 of BAUDCTL register
        bsf   TXSTA ^0x80, BRGH   ; BRGH set HI
        bcf   TXSTA ^0x80, SYNC   ; SYNC LO , enable serial port asych.
        bcf   TXSTA ^0x80, TX9          ; 8bit data
        bsf   TXSTA ^0x80, TXEN         ; enable Transmiter

        bcf   STATUS,RP0           ; back to Register Page 0
        bsf   RCSTA, SPEN   ; enable serial port
        bcf   RCSTA, CREN   ; continuous recv. disable
        bsf   RCSTA, RX9          ; 9 bit parity
        movf  RCREG, w             ; read RECREG to clear any pending IRQ

;**************************     Begin  Main   Loop ***********************
StartPosition
       movlw  0x05               ; load the starting position variable = 5
       movwf  Position           ; position 5 is 6 degrees from far R. limit
MoveServos                       ; test if servo is moving to position 5
       nop                       ; if yes, allow more time for servo to move
       bcf    STATUS,Z           ; the bigger distance before firing off sonar.
       movf   Position,w         ; Position to WREG
       xorlw  0x05               ; Is Position == 5 ?
       btfss  STATUS,Z
       goto   SetDelay3          ; if NOT, then skip to below
       movlw  0x09               ; want 9 * 20 ms delays = 180 ms
       goto   SaveDelay          ; for servo to move 72 degrees
SetDelay3
       movlw  0x03               ; Only moving 18 degrees. Three 20 ms delays
SaveDelay                        ; before any ranging attempt. 3* 20ms = 60 ms
       movwf  PulseCount         ; of 'dead' cycles with no sonar pinging

SendSignalsAndWait
       call   SubSendServoPulse  ; send out HI pulse on the servo control pin
                                 ; duration of HI pulse depends on Position
       call   SubDEL20           ; do a 20 ms delay
       decfsz PulseCount,f       ; Count down from 3 to zero, then begin
```

```
       goto   SendSignalsAndWait  ; ranging with sonars, else do another
                                  ; 'dead' cycle with no sonar firing.

                              ; Do ranging with each of 4 sonars, one at a time
       movlw  0x4             ; load counter with initial value = 4
       movwf  SonarNum        ; to keep track of which sonar is being
                              ; pinged.
RangingAttempt
       nop
       call   SubSendServoPulse   ; must send out a pulse to the servo first
       nop                        ; then we have 20 to 30 ms to use for sonar
                                  ; ranging attempt before the servo needs
                                  ; another signal

       ; case switch logic to pick which sonar to send INIT pin HI out
       movf   SonarNum,w
       xorlw  4
       btfss  STATUS,Z
       goto   NotSonar4
       PingSonar4Mac
       goto   PingSent
NotSonar4
       xorlw  4^3
       btfss  STATUS,Z
       goto   NotSonar3
       PingSonar3Mac
       goto   PingSent
NotSonar3
       xorlw  3^2
       btfss  STATUS,Z
       goto   NotSonar2
       PingSonar2Mac
       goto   PingSent
NotSonar2
       PingSonar1Mac

PingSent
; set up Capture Compare to catch rising edge of echo
       clrf   CCP1CON
       movlw  b'00000101'         ; capture the RISING edge of echo signal
       movwf  CCP1CON             ; when it's passed through XOR-gate

       bcf    T1CON,0             ; TIMER1 disable
       clrf   PIR1                ; clear TMR1 overflow flag
       clrf   TMR1H               ; and Timer1 capture flag
       clrf   CCPR1H              ; clear high and low bytes of counters
       clrf   TMR1L               ; clear the capture compare values, too
       clrf   CCPR1L              ; before next ranging
;********************** Ping, sent  start timing **********************
       bsf          T1CON,0       ; enable TIMER1

                                  ; delay 1ms for transducer to stop ringing
OneMsDelay
       movlw        0x04   ; do 4 delays
       movwf        j
       movlw        0xA7   ; 167* 3 instructions/loop * 0.5 usec = 250 usec
       movwf        i      ; delay to allow for sonar transducer ring down
       decfsz       i,f    ; and allow time for the "crud" and "
                           ; "glitches on the echo
       goto   $-1          ; line to go away
       decfsz j,f
       goto   $-5
```

```
       ; send blanking inhibit to override the default 2.38 ms blanking
       movf   SonarNum,w
       xorlw  3            ; is it blanking inhibit for sonar 3?
       btfss  STATUS,Z
       goto   NotBINHforSonar3
       movlw  b'00110000'  ; send this value out PORTA if it's sonar3
       movwf  PORTA
       goto   BINHsent
NotBINHforSonar3
       xorlw  3^1
       btfss  STATUS,Z     ; it's not sonar 3.  Is it sonar1?
       goto   NotBINHforSonar1
       movlw  b'00010100'  ; send BINH pin HI and keep the sonar INIT high
       movwf  PORTA        ; for sonar 3
       goto   BINHsent
NotBINHforSonar1
       movlw  b'00010000'  ; Not sonars 1 or 3.  Must be 2 or 4.
       movwf  PORTA        ; regardless, we're not pinging a sonar in PORTA
                           ; so just sent the BINH pin in PORTA hi.
BINHsent
       nop                 ; done
       clrf   PIR1         ; clear TMR1 overflow flag.
                           ;MAKE SURE CAPTURE  FLAG IS CLEAR
                           ; AFTER THE BINH SIGNAL GETS SENT.

check_echo
       btfsc  PIR1,2       ; check for capture of rising edge of echo
       goto   check_done   ; if capture flag is HI, then get out of loop
       btfss  PIR1,0       ; check for over flow of TIMER1 count
       goto   check_echo   ; if overvlow flag not set, then go back up and do
                           ; again
       goto   ovr_flo      ; TIMER1 must have overflowed


check_done
       bcf    T1CON, TMR1ON       ; stop  TIMER1 counter
       movf   CCPR1L,w            ; copy the values of capture compare
       movwf  SonarRange          ; counters to vars.  Keep 7 bits of high byte
                                  ; and MSB of LO byte.
       movf   CCPR1H,w            ; copy high byte of the couter to high byte of
                                  ; range variable
       movwf  SonarRange+1
       btfsc  SonarRange+1,7      ; test the MSB of HI byte.  See if value will
                                  ; overflow
                                  ; 1 byte output.
       goto   ovr_flo      ; if set, then value is too big to represent it
                           ; with 1 byte.
       rlf    SonarRange,w        ; shift 7th bit out of LSByte and into carry
                                  ; Flag C
       rlf    SonarRange+1,f      ; shift the carry flag value into bit 0 of
                                  ; MSByte and
                                  ; shift MSBit out of register to
                                  ; the carry Flag C
                                  ; i.e. throw out the 2^16th counter value.
                                  ; Keep 2^8 thru 2^15 bits.

       bsf    T1CON, TMR1ON       ; restart TIMER1 counter
       bcf    PIR1,2              ; reset the capture compare flag
       btfss  PIR1,0              ; continue timing to the end of normal
       goto   $-1                 ; listening period so that next servo
                                  ; pulse will be about 20 to 30 ms after
                                  ; RangingAttempt
```

```
       goto   done

ovr_flo
       movlw  0x02
       movwf  SonarRange          ; set equal to 2 for no contact
       movwf  SonarRange +1       ; set equal to 0x02 for no contact flag

;********** ********** Done with Timing **********************
done
       movf   SonarRange+1,w      ; do sanity check on the 1 byte range value
       sublw  0xFA                ; subtract WREG(SonarRange)
                                  ; from literal 250, result to WREG
       btfsc  STATUS,C            ; test carry flag. Is SonarRange+1 byte > 250?
       goto   saveValueToArr      ; skip ahead and save value to array
       movlw  0x02                ; else...
       movwf  SonarRange+1        ; if value 251 to 255, report as no
                                  ; contact
saveValueToArr
                   ; store the range value in a unique spot in the array
                   ; Put sonar reading in an array location depending on which
                   ; sonar fired and which position it was pointed at.
                   ; the 2^3 and 2^2 bits indicate the position
                   ; the 2^1 and 2^0 bits indicate the sonar number (front,
                   ; rear, L, R)
                   ; array index = 4*(5 - Position) + (4 - SonarNum)
       movf   Position,w          ; load WREG with current Position
       sublw  5                   ; find difference 5 - Position put into WREG
       movwf  ArrIndex            ; make a copy put into ArrIndex
       bcf           STATUS,C     ; carry value = 0
       rlf           ArrIndex,f   ; shift value to left.  i.e. multiply by 2
       bcf    STATUS,C            ; carry value = 0
       rlf           ArrIndex,f   ; shift 0 into bit 0 and others over to the
                                  ; left. i.e. multipy by 2 again

       movf   SonarNum,w          ; load WREG with sonar number that was pinged
       sublw  4                   ; find 4 - SonarNum and put result in WREG
       addwf  ArrIndex,f          ; sum WREG + ArrIndex = ArrIndex
                   ; store high byte in SonarRanges array at ArrIndex location
       movf   ArrIndex,w
       addlw  SonarRanges
       movwf  FSR           ; Set FSR to the array index offset + SonarRanges
                            ; file register location
       movf   SonarRange+1,w      ; want to save the High byte of SonarRange
       movwf  INDF                ; move WREG to the array position

       clrf   PORTA         ; send all INIT pins and the BINH pin low
       clrf   PORTB

       nop

NextNum
       decfsz SonarNum,f          ; Done with delay.  Decrement through sonars 4
to 1
       goto   RangingAttempt      ; do a sonar ranging "ping" for next sonar.
                                  ; Same azimuth position.

       decfsz Position,f    ; After all 4 sonars fired, decrement position.

       goto   MoveServos    ; Go back up to start of loop and move servos to
                            ; new position.
```

```
                    ; if Position == 0, then we've fired off
                    ; at positions 5,4,3,2,1. Need to reset to position to
                    ; 5.
       movlw  0x05              ; Reset position to position 5, start position
       movwf  Position
       goto   StartPosition     ; Repeat loop and have servo motor reset to
                                ; the start position = far Right.

; ******************** Subroutines *****************************
; Subroutine to send the RC servo pulse to the Futaba 3003 servo
SubSendServoPulse
       clrf   CCP1CON           ; turn off the ccp1 module
                                ; set up TIMER0
       bsf    STATUS,RP0        ; Bank 1
       movlw  b'00000011'       ; Sourced from the Processor clock
       movwf  OPTION_REG    ; Bits 2,1,0 are 011, so prescaler 1:16 or
                                ; 8 usec per bit for Servo Pulse HI time
       bcf    STATUS,RP0        ; bank 0 again
       bcf    STATUS,Z      ; clear this in case it's set, want to check against
                            ; it later
       bcf    INTCON,T0IF   ; clear the overflow flag and then...
       clrf   TMR0          ; zero out TMR0 count
                            ; case switch logic to find out which position
       movf   Position,w    ; use Position var. to determine duration of
                            ; HI pulse to servos.
                            ; Position in degrees from full Left =
                            ;Case switch logic
                            ; test == 5?
       xorlw  5
       btfss  STATUS,Z
       goto   NotPosit5
       servoHiMac    ; pulse the Servo pin HI for 105 "ticks" of 8usec each
       movlw  0x97          ; count up from decimal 151 to 256
       movwf  TMR0          ; to keep pulse HI for 840 usec for 54 degree posit
       btfss  INTCON,T0IF
       goto   $-1
       bcf    INTCON,T0IF   ; clear the overflow flag and then ...

       goto   PulseDone


NotPosit5
       xorlw  5^4               ; test == 4?
       btfss  STATUS,Z
       goto   NotPosit4
       servo  HiMac             ; pulse the Servo pin HI count 127 ticks
       movlw  0x80              ; count up from 256 - 128.5 = 128 decimal
       movwf  TMR0              ; to keep pulse HI for 1020 usec
       btfss  INTCON,T0IF       ; for 72 degree position
       goto   $-1
       bcf    INTCON,T0IF       ; clear the overflow flag and then ...
       nop
       nop
       nop
       nop                  ; add a few nop cycles to get just about 1020 usec
       goto   PulseDone

NotPosit4
       xorlw  4^3           ; test == 3?
       btfss  STATUS,Z
       goto   NotPosit3     ; skip below code if Position is NOT 3
                            ; DO this code if Position == 3 center = 90 deg
```

```
                        ; from far R. limit
       servoHiMac       ; pulse the Servo pin HI for 150 ticks
       movlw  0x6A      ; count up from 256 - 150 = 106 to 256
       movwf  TMR0      ; to keep pulse HI for 1200 usec
       btfss  INTCON,T0IF
       goto   $-1
       bcf    INTCON,T0IF  ; clear the overflow flag and then ...

       goto   PulseDone

NotPosit3
       xorlw  3^2
       btfss  STATUS,Z
       goto   NotPosit2
                    ; DO this code if Position == 2 == 18 degrees L of neutral
                    ; or 108 degrees from far R. limit
       servoHiMac       ; pulse the Servo pin HI for 172.5 ticks
       movlw  0x54      ; count up from 256 - 172.5 = 83.5 = 84
       movwf  TMR0      ; to keep pulse HI for 1380 usec
       btfss  INTCON,T0IF
       goto   $-1
       bcf    INTCON,T0IF  ; clear the overflow flag and then...
       nop
       nop
       nop    ; add a couple of nop cycles to add a bit more time to servohi

       goto   PulseDone
NotPosit2           ; Must be Position == 1 == 126 degrees from Far R. limit
       servoHiMac       ; pulse the Servo pin HI for 1560/8 = 195 ticks
       movlw  0x3D      ; count up from from 256 - 195 = 61
       movwf  TMR0      ; to keep pulse HI for 1560 usec
       btfss  INTCON,T0IF
       goto   $-1
       bcf    INTCON,T0IF  ; clear the overflow flag and then...
       nop
       nop
       nop
       nop
PulseDone
       servoLoMac       ; set the servo to low again
       nop
       return



; delays for 20 ms using TIMER0.  Sends Data via RS232.
; Note prescaler change to 1:256.
SubDEL20
       bsf    STATUS,RP0          ; Bank 1
       movlw  b'00000111'         ; configure Timer0.
       movwf  OPTION_REG          ; Prescaler 1:256 so 1 tick = 128 usec
       bcf    STATUS,RP0          ; Bank 0
       movlw  0x64                ; timer0 count from 100 to 256
       movwf  TMR0                ; so count 156 ticks = 19.97 ms
       btfss  INTCON,T0IF         ; wait for overflow
       goto   $-1
       bcf    INTCON,T0IF
       bsf    STATUS,RP0          ; Bank 1
       movlw  b'00000011'         ; Reconfigure TMR0 back to 1:16
       movwf  OPTION_REG
       bcf    STATUS,RP0          ; Bank 0
                                  ; send out the range data to serial
```

```
      bcf    PIR1,TXIF           ; clear bit
                                 ; load WREG with literal value of 0xFE
      movlw  0xFE                ; load WREG with flag to signal start of
                                 ; sonar sentence
      movwf  TXREG               ; send the sonar range byte out serial
      nop
      nop
      btfss  PIR1,TXIF
      goto   $-1
      bcf    PIR1,TXIF           ; clear bit

                         ; repeat start flag
      movlw  0xFE        ; load WREG with flag to signal start of
                         ; sonar sentence
      movwf  TXREG       ; send the sonar range byte out serial
      nop
      nop
      btfss  PIR1,TXIF
      goto   $-1
      bcf    PIR1,TXIF    ; clear bit

; BEGIN data section of serial transmission
; Very Kludge.  Could implement a counter and loop
; to access the array values.
; Data output begins with value that is closest to yaxis of robot
; and proceeds clockwise from 6degrees to 24, to 42, etc. to last
; value (array element 6) which is 348 degrees
                                       ; element 2
      movf   SonarRanges+0x02,w  ; load WREG with 1 byte
      movwf  TXREG       ; send the sonar range byte out serial
      nop
      nop
      btfss  PIR1,TXIF
      goto   $-1
      nop
      bcf    PIR1,TXIF           ; clear bit

                                       ; element 19
      movf   SonarRanges+0x13,w  ; load WREG with 1 byte
      movwf  TXREG               ;send the sonar range byte out serial
      nop
      nop
      btfss  PIR1,TXIF
      goto   $-1
      nop
      bcf    PIR1,TXIF           ; clear bit

                                 ; element 15
      movf   SonarRanges+0x0F,w  ; load WREG with 1 byte
      movwf  TXREG        send the sonar range byte out serial
      nop
      nop
      btfss  PIR1,TXIF
      goto   $-1
      nop
      bcf     PIR1,TXIF          ; clear bit


                                 ; element 11
      movf   SonarRanges+0x0B,w  ; load WREG with 1 byte
      movwf  TXREG               ; send the sonar range byte out serial
      nop
```

```
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 7
movf   SonarRanges+0x07,w  ; load WREG with 1 byte
movwf  TXREG               ;send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 3
movf   SonarRanges+0x03,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit


                           ; element 16
movf   SonarRanges+0x10,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit
                           ; element 12
movf   SonarRanges+0x0C,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 8
movf   SonarRanges+0x08,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 4
movf   SonarRanges+0x04,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
```

```
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit


                           ; element 0
movf   SonarRanges,w       ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 17
movf   SonarRanges+0x11,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 13
movf   SonarRanges+0x0D,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 9
movf   SonarRanges+0x09,w  ; load WREG with 1 byte
movwf  TXREG         ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 5
movf   SonarRanges+0x05,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 1
movf   SonarRanges+0x01,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
```

```
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 18
movf   SonarRanges+0x12,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 14
movf   SonarRanges+0x0E,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 10
movf   SonarRanges+0x0A,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; element 6
movf   SonarRanges+0x06,w  ; load WREG with 1 byte
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit
                           ; indicate which byte is newest
                           ; i.e. time late = 0.
movf   ArrIndex,w          ; load WREG with 1 byte
movwf  TXREG               ; send the last byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
nop
bcf    PIR1,TXIF           ; clear bit

                           ; send end flag
movlw  0xFB                ; load WREG with flag to signal end of
                           ; sonar sentence
movwf  TXREG               ; send the sonar range byte out serial
nop
nop
btfss  PIR1,TXIF
goto   $-1
bcf    PIR1,TXIF           ; clear bit
```

```
return
 end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F - DYNAMIC C ROBOT OPERATING CODE

```
////////////////////////////////////////////////////////////////////////////
//creature6mod2.c
//15NOV2007
// Dynamic C 9.21 running on BL2600
// ver5 adds IMU support and heading hold from IMU
// ver6 turns on random sonar walk if contact within 40 cm
// added IR speed sensitive threshold
// added I2C wheel counter dr
// added magnetic variation correction.  dr with respect to TRUE NORTH.

////////////////////////////////////////////////////////////////////////////
// Compiler settings for comms module test and for debugging with printf
////////////////////////////////////////////////////////////////////////////
//set MODULE_TEST to 1 when running as a separate module
//set MODULE_TEST to 0 when integrate with other files (using #use " ")
#define  MODULE_TEST    0
//set DEBEGPRINT to 1 to enable all printf stdio for debugging
//set DEBUGPRINT to 0 to disable printf when robot not connected to laptop
#define DEBUGPRINT      0

//set DEBUGSONAR to 1 to enable ONLY printf stdio for debugging SONAR STUFF
//set DEBUGSONAR to 0 to disable printf
#define DEBUGSONAR      1
//set to enable printf statements to display wheel rotation sensors,
//dr data, etc
#define DEBUGDR         1

////////////////////////////////////////////////////////////////////////////
//Declarations, global variables
////////////////////////////////////////////////////////////////////////////
#class auto
void msDelay(unsigned int delay);
////////////////////////////////////////////////////////////////////////////
// 2. Communications Module Code
////////////////////////////////////////////////////////////////////////////
/*       CommModule.c
        Charles Le, Zach Cole, and John Gamble

        This module enable the communication between the BL2000 and the remote
        control station (Java program) via UDP.  There are total of 5 channels
        that UDP packets are sent to:
        4001 -  MAN_CTRL_PORT packets (from Remote Control to BL2000)
        4002 -  WYPNT_PORT - Way point packets (from Remote Control to BL2000)
        4003 -  GPS_PORT - GPS data (from BL2000 to Remote Control)
        4004 -  COMPASS_PORT - Compass data (BL2000 to Remote Control)
        4005 -  ERROR_PORT - Error info (BL2000 to Remote Control)

*******************************************************************************
/
/*
 * Pick the predefined TCP/IP configuration for this sample.  See
 * LIB\TCPIP\TCP_CONFIG.LIB for instructions on how to set the
 * configuration.
 */
#define TCPCONFIG 1

/*
```

```
 * Define the number of socket buffers that will be allocated for
 * UDP sockets.  We need five UDP sockets, so five socket buffer
 * will be used for MAX_UDP_SOCKET_BUFFERS.
 * FUTURE WORK: should only use one udp port for communication.
 * Concatinate all informations to one packet, and detokenize when it arrives
 *     destination.
 */
#define MAX_UDP_SOCKET_BUFFERS 5

//The following udp ports are set up to communicate with the ctrl station
#define MAN_CTRL_PORT   4001       //sent from Ctrl station (input port)
#define WYPNT_PORT      4002       //sent from Ctrl station (input port)
#define GPS_PORT        4003       //sent to Ctrl station (output port)
#define COMPASS_PORT    4004       //sent to Ctrl station (output port)
#define ERROR_PORT      4005       //sent to Ctrl station (output port)

//These are constants used in the Communication Module
#define      UDP_BUFF_SIZE  256
#define      TRUE                    1
#define      FALSE                   0
#define      PRESENT                 1
#define      NOT_PRESENT         0

//These constants specified the type of navigation data sent to the ctrl module
#define      GPS_DATA                1
#define      COMPASS_DATA 2
#define      ERROR_DATA          3

/*
 * If this is set to "0", we will accept a connection from anybody.
 * The first person to connect to us will complete the socket with
 * their IP address and port number, and the local socket will be
 * limited to that host only.
 *
 * If it is set to all "255"s, we will receive all broadcast
 * packets instead.
 *
 *     THIS ADDRESS NEED TO BE IN THE SAME SUBNET WITH THE LOCAL IP ADDRESS
 */
#define REMOTE_IP       "192.168.4.37"
//#define REMOTE_IP     "255.255.255.255" /*broadcast*/

/*******************************
 * End of configuration section *
 *******************************/

#memmap xmem
#use "dcrtcp.lib"
#use "udp.lib"

//This data structure describe the data structure of the communication
//between the Communication Module and the Navigation Module
typedef struct CommChanStruct
{
       int DataPresentFlag;
   char Buff[UDP_BUFF_SIZE];
} CommChan;

int receive_packet(void);
int receive_packet_from_port(int );
int send_packet(int PortNumber, char* );
void ClearDataPresent(CommChan *);
```

```
void SetDataPresent(CommChan *);
int CheckDataPresent(CommChan *);
void ClearChannelBuffer(CommChan *);
void InitUdpComm(void);
int SendToControlStation(int );
void SendGpsToControlStation(void);
void SendCompassToControlStation(void);
void SendErrorToControlStation(void);
void SendToCommModule(CommChan *, char *);
char* GetMessageFromCommModule(CommChan *);

//These are udp sockets that communicate with the ctrl station
static udp_Socket ManCtrlSock, WayPtSock, GpsSock, CompassSock, ErrorSock;

//These are the communication channels between the Communication Module
//and the Navigation Module
static CommChan ManCtrlChan, WayPtChan,GpsChan, CompassChan, ErrorChan;
char ErrString[200]; //for feedback to laptop control station


////////////////////////////////////////////////////////////////////////////
// 3. IMU
////////////////////////////////////////////////////////////////////////////

// serial buffer size
#define FINBUFSIZE  255
#define FOUTBUFSIZE 31
// serial baud rate
#define BAUD232 19200
#define IMUHEADLIMIT 0.0175
#define IMUROTLIMIT 0.010227
char imuChar;
char *xRot, *yRot, *zRot; // raw x, y, and z axis body rotation chars from IMU
int imuZeroPoint;    //nominal 511 for zero rotation
float zRotation[7]; // zRot in float + or -
                                                  // value in rad/s rotation rate
of chassis
long  imuTimes[7];   //hold the times the IMU was recorded
float imuHeading;
int   newImu;   //boolean flag new IMU data avail
int    getImu(void);     // reads IMU sentence and parses for z axis rotation
rate


////////////////////////////////////////////////////////////////////////////
// 4. I2C Compass
////////////////////////////////////////////////////////////////////////////

#use "I2C.LIB"
//reads I2C compass sets global 1 byte integer heading
int compass(char addrsByte, char registerByte);

//PA6 SCL output, PA7 SDA output, PB0 SCL input PB2 SDA input
//Definition of Clock and Data Ports
// Enable Digital out DIO 00 and DIO 01 digital outputs (sinking only)
// Enable Digital DIO 2,03,04 outputs for motor direction signals
#define DIGOUTCONFIG       0x001F
int heading;        // integer 1 to 255 value current observed mag heading
int headingLast;      // last observed heading from compass
int newCompass;   // boolean flag set when new compass data is available
                                       // from I2C compass
int cmdHeading;      // heading commanded by CPU to maintain
```

```
int holdHeading;     // boolean TRUE if heading hold engaged
//MAGVAR Monterey 14 E as 1 byte value (14 deg * 255 bits/360 deg)
#define MAGVAR 9.91667
#define NORTH 255    // 1 byte value is 255 for 360 degrees
// limit of allowed hdg error 2 * 1.4 degrees = 2.8 degree slop
#define HEADINGSLOP 2
//Proportional coefficient for PID
#define KAPPA_P     12
//Derivative coeff. for PID heading control feedback
#define KAPPA_D     0.8
#define KAPPA_I     0


////////////////////////////////////////////////////////////////////////////
// 5. Waypoint
////////////////////////////////////////////////////////////////////////////
typedef struct
{
       double lat;
       double lon;
       char action;
}WP;

WP waypoints[13];          // 0th element is the Origin lat, long.
                                                    // Then 1 to 12th elements
are waypoints
int current_wp_count;   // a counter to keep track of waypoints

void getwaypoints(void);   // gets waypoints from GUI  calls initNav()
                                                          //   to   initialize
nav
void makeWayPts(void);            // fills in array of type Position

////////////////////////////////////////////////////////////////////////////
// 6. Navigation
////////////////////////////////////////////////////////////////////////////
#use "gps.lib"
// POSITACCURACY IN cm
#define POSITACCURACY 200

// uses flat Earth approximation around robot's current position
// Define polarity for robot,  N hemisphere + and East hemisphere +
//e^2 = f*(2-f) where f = 1/298.257223563 for WGS84
const float esquared = .0066943800;
const float a = 6378.13700; //radius Earth in km for WGS84
//meridional radius of curvature and radius of curvature for prime vertical km
float R1, R2;
GPSPosition OriginPos; // position of the flat Earth ORIGIN
double initPos_lat, initPos_long;  // lat, long in radians of the Origin
char fakeGPSsentence[64];  //holds a fake GGA sentence to pass to GUI
typedef struct{
       long cm_N;
   long cm_E;
} Position;
Position currentPos;  // declare a var of type Position for current robot pos.
Position DRpos;       // position based just on DR from wheel data
Position waypointsPos[13]; // array of Position type to hold x, y
                                                          //   loctations   of
waypoints

void getOriginGPSfromGUI(void);
void initNav(void);
```

```
void makeFakeGPSpos(void);
void navigation(int lastWaypt, int nextWaypt);

////////////////////////////////////////////////////////////////////////
// 7. DR
////////////////////////////////////////////////////////////////////////

int wheel1, wheel2, wheel3;             // observed values from wheel counter
in Hz
                                                                        //
optical wheel sensor produces 50 Hz in 1 rev
                                // or 50 Hz in 2*pi radians
float obsOmega1, obsOmega2, obsOmega3;  // observed radians/s wheel speeds
float obsVx, obsVy;                     // observed robot velocity WRT robot's
y, x axis cm/s
float vE[3];                            // velocities in Earth Frame in cm/s
float vN[3];
float obsChassisOmega;  // obsv'd robot chassis rotational speed rad/s
float obsTheta;                  // obsv'd robot velocity vector direction
                                                // WRT robot's y axis

float obsV;                             // magnitude of robot's obsv'd velocity
vector
int newDR;                              //  boolean  flag  set  if  newDR  wheel
speed data avail
long DRtime;                     //  time  elapsed  since  last  DR.    Multipy
velocities
                                                // times the DRtime to get
position estimate
//reads 1 wheel speed from counter
int wheel(char addrsByte, char registerByte);
// calculates velocity vector from wheel speeds in rad/s
void calcVeloRobotFrame(float omega1, float omega2, float omega3);

////////////////////////////////////////////////////////////////////////
// 8. Obstacle Avoidance     (IRs)
////////////////////////////////////////////////////////////////////////

float irRanges[6]; // 6 element array to hold the current IR voltage readings
float irRangesOld1[6];  // array for the IR data previous to current data
float irRangesOld2[6];  // array for IR data prior to Old1 (oldest data)
float irRangesAvg[6];      // average values of last two IR data sets
long watchDogTime;    //time last OK motion.  For detection stuck wheels
#define TIMELIM    4000
#define STUCKLIMIT 0.1
#define IRSTOP_RNG              1.6
#define IRCAPTURE_RNG           1.0
#define CRAWLSPEED              .5
#define SLOWROTATESPEED         3
int   irCloseContact(void);     //tests   irRanges[]   for   readings   exceeding
IR_STOP_RNG

////////////////////////////////////////////////////////////////////////
// 9. Sonar Sensor
////////////////////////////////////////////////////////////////////////

//serial C port to PIC
#define SONARCONVERSION 1.05
#define CINBUFSIZE  127
#define COUTBUFSIZE 63
#define CLOSERANGE  40
#define MIDRANGE    90
```

```
#define NOCONTACT   250
#define NUM_SONAR_AZIMUTHS 20    // 20 azimuths with 2 bytes per range reading
char input_char;                 // serial char input
int avoidObstacles;     // boolean flag True of False to do obstacle avoidance
float sonarRanges[24];     // array to hold range data in cm for 20 azimuths
int newestSonar;           //holds number of the newest sonar reporting range
int closeContact;          // boolean flag sonar contact close
int objectAhead;           // boolean flag obstacle ahead
int clutterdArea;          // boolean flag for behavior
int getSonarRanges(void);
int checkSonarRanges(void);
long doRandSpin(void);           // does random spin in place for 500 to 1500
ms
long spinDelay;         // time in ms to delay while spinning in place

////////////////////////////////////////////////////////////////////////////
// 10. Motion
////////////////////////////////////////////////////////////////////////////

#define RADWHEEL    5.250  // radius in cm for motion in cm
#define RADCHASSIS  22    // radius from bot's center to wheel cm
#define FWD_DIR 0
#define REV_DIR 1
#define STOPVOLTS 0.00
#define MAXVOLTS 4.10
// max angular wheel speed
// approx 220Hz on tach (with 50 division/1 rev optical targets)
// or 27.6 radians /s
//#define MAXOMEGA  13.80    WRONG VALUE
#define MAXOMEGA  27.64
//analog speed signals output
#define mtr1Chan 0
#define mtr2Chan 1
#define mtr3Chan 2
// channels for digital direction signals
#define dir1Chan 2
#define dir2Chan 3
#define dir3Chan 4

// motion, hdg vars and function prototypes
int robotStopped;  //boolean flag TRUE if wheel counters go to zero
int robotNotRotating;  //boolean flag set TRUE if chassisOmega zero
float mtr1Spd;  //motor 1 analog voltage speed signal 0 to 4.096 (from BL2000)
float mtr3Spd; // motor 2 analog speed signal
float mtr2Spd; // motor 3 analog speed signal
int dir1;      // motor 1 direction.
                                 // boolean values 1 = reverse = REV_DIR, 0 =
FWD_DIR
int dir3;      // motor 2 direction
int dir2;      // motor 3 direction
float v;         //robot's velocity vector magnitude, using scale 0 to 1.0
float cmdVelocity;   // remember original velocity to maintain
float theta;        // angle in degrees between Y axis (alligned with motor 1)
                                       // and vector
float chassisOmega;  //body or chassis rotation angular velocity rad/s

// function prototypes
int stopMotors(void);    // stops motors.  calls setMotorVolts
int spinInPlace(void);   // rotate in place
int getVeloVect(void);         //gets user input
int solveMotorSpeeds(void);  //solves three wheel speeds given v and theta
// converts a wheel angular speed in rad/s to analog volts needed to output
```

```
float freqToVolts(float omegaIn);
// sets analog output values to pins
int setMotorVolts(float spd1, int dir1, float spd2, int dir2, float spd3,\
       int dir3);
int vector(void);       //move robot in any direction calls setMotorVolts


////////////////////////////////////////////////////////////////////////
// 11. Control
////////////////////////////////////////////////////////////////////////
int manual_control_flag;    // boolean flag indicates user has taken manual
control
void manual_control(void);

////////////////////////////////////////////////////////////////////////
// Shared functions                                                   //
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Function: msDelay
// Programmer: Kirk Volland
// Input: desired time delay in ms
// Output: Nil
// Description: To delay the processor with the input time in ms, primarily
// for serial and I2C communications.
////////////////////////////////////////////////////////////////////////
void msDelay(unsigned int delay)
{
       unsigned long done_time;

       done_time = MS_TIMER + delay;
   while( (long) (MS_TIMER - done_time) < 0 );
}

////////////////////////////////////////////////////////////////////////
// END GLOBAL VARS DECLARATIONS, MACROS, AND FUNCTION PROTOTYPES
// BEGIN FUNCTION DEFINITIONS
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//2. Communications
////////////////////////////////////////////////////////////////////////
/

////////////////////////////////////////////////////////////////////////
/
// Function: receive_packet
// Programmer: Charles Le
// Input: None
// Output: return 1 for a successful receive, and 0 if fail to send
// Description: This program will check for way point input and manual control
//            input from the remote control station (Java Program)
////////////////////////////////////////////////////////////////////////
/
int receive_packet(void)
{

   #GLOBAL_INIT
   {
      memset(ManCtrlChan.Buff, 0, sizeof(ManCtrlChan.Buff));
      memset(WayPtChan.Buff, 0, sizeof(WayPtChan.Buff));
   }
```

```
   memset(ManCtrlChan.Buff, 0, sizeof(ManCtrlChan.Buff));
   memset(WayPtChan.Buff, 0, sizeof(WayPtChan.Buff));

   udp_recv(&WayPtSock, WayPtChan.Buff, sizeof(WayPtChan.Buff));
  #if DEBUGPRINT
  printf("\n WayPtSock-> %s\n",WayPtChan.Buff);
  #endif

   // receive the packet
   udp_recv(&ManCtrlSock, ManCtrlChan.Buff, sizeof(ManCtrlChan.Buff));

  #if DEBUGPRINT
  //printf("\n ManCtrlSock-> %s\n",ManCtrlChan.Buff);
  #endif

   tcp_tick(NULL);
   return 1;
}
////////////////////////////////////////////////////////////////////////////
/
// Function: receive_packet_from_port
// Programmer: Charles Le
// Input: None
// Output: return 1 for a successful receive, and 0 if fail to send
// Description: This program will receive udp packet from the specified port
////////////////////////////////////////////////////////////////////////////
/
int receive_packet_from_port(int PortNumber)
{
   int retval;

   #GLOBAL_INIT
   {
      memset(ManCtrlChan.Buff, 0, sizeof(ManCtrlChan.Buff));
      memset(WayPtChan.Buff, 0, sizeof(WayPtChan.Buff));
   }

   memset(ManCtrlChan.Buff, 0, sizeof(ManCtrlChan.Buff));
   memset(WayPtChan.Buff, 0, sizeof(WayPtChan.Buff));

   switch(PortNumber)
   {
       case WYPNT_PORT:
      {
       retval = udp_recv(&WayPtSock, WayPtChan.Buff, sizeof(WayPtChan.Buff));
         if (retval < 0) {
                   sock_close(&WayPtSock);
             if(!udp_open(&WayPtSock, PortNumber, /*resolve(REMOTE_IP)*/ -1, \
             PortNumber, NULL))
            {
                   exit(0);
             }
             }
         else
         {
            SetDataPresent(&WayPtChan);
         }
         tcp_tick(NULL);
      }
      case MAN_CTRL_PORT:
      {
```

```
       retval = udp_recv(&ManCtrlSock, ManCtrlChan.Buff, \
              sizeof(ManCtrlChan.Buff));

         if (retval < 0) {
                  sock_close(&ManCtrlSock);
              if(!udp_open(&ManCtrlSock,  PortNumber,  /*resolve(REMOTE_IP)*/  -1,
\
              PortNumber, NULL))
             {
                    exit(0);
              }
              }
         else
         {
            SetDataPresent(&ManCtrlChan);
         }
         tcp_tick(NULL);
       }
       default:
       {
        //error incorrect port number
          return 0;
       }

   }
   tcp_tick(NULL);
   return 1;
}
////////////////////////////////////////////////////////////////////////////
/
// Function: send_packet
// Programmer: Charles Le
// Input: PortNumber, messageIn
// Output: Return 1 for a successful sending and 0 for fail to send
// Description: The function will send the input message to the input udp port
////////////////////////////////////////////////////////////////////////////
/
int send_packet(int PortNumber, char* messageIn)
{
   auto int    length, retval;
   udp_Socket *sock;

    length = strlen(messageIn) + 1;

   switch(PortNumber)
   {
       case GPS_PORT:
       {
       retval = udp_send(&GpsSock, messageIn, length);
         if (retval < 0) {
       #if DEBUGPRINT
              printf("Error   sending   datagram!      Closing   and   reopening
socket...\n");
       #endif
                  sock_close(&GpsSock);
              if(!udp_open(&GpsSock, PortNumber, /*resolve(REMOTE_IP)*/ -1, \
              PortNumber, NULL))
             {
                    #if DEBUGPRINT
                           printf("udp_open failed!\n");
                    #endif
                    exit(0);
```

```
            }
            }
      }
      case COMPASS_PORT:
      {
       retval = udp_send(&CompassSock, messageIn, length);
          if (retval < 0) {
       #if DEBUGPRINT
           printf("Error sending datagram!  Closing and reopening socket...\n");
       #endif
                  sock_close(&CompassSock);
            if(!udp_open(&CompassSock,  PortNumber,  /*resolve(REMOTE_IP)*/  -1,
\
            PortNumber, NULL))
           {
            #if DEBUGPRINT
            printf("udp_open failed!\n");
            #endif
            exit(0);
            }
            }
      }
      case ERROR_PORT:
      {
       retval = udp_send(&ErrorSock, messageIn, length);
          if (retval < 0) {
       #if DEBUGPRINT
           printf("Error sending datagram!  Closing and reopening socket...\n");
       #endif
                  sock_close(&ErrorSock);
            if(!udp_open(&ErrorSock, PortNumber, /*resolve(REMOTE_IP)*/ -1,\
             PortNumber, NULL))
           {
            #if DEBUGPRINT
            printf("udp_open failed!\n");
            #endif
            exit(0);
            }
            }
      }
      default:
      {
       //error incorrect port number
          return 0;
      }

   }
   tcp_tick(NULL);
   return 1;
}
////////////////////////////////////////////////////////////////////////////
/
// Function: ClearDataPresent
//     Programmer: Charles Le
// Input: DataChannel
// Output: None
// Description: This function will set DataPresentFlag to 0
////////////////////////////////////////////////////////////////////////////
/

void ClearDataPresent(CommChan *Channel)
{
```

```
	Channel->DataPresentFlag = FALSE;
}
////////////////////////////////////////////////////////////////////////////
/
// Function: SetDataPresent
//     Programmer: Charles Le
// Input: DataChannel
// Output: None
// Description: This function will set DataPresentFlag to 1
////////////////////////////////////////////////////////////////////////////
/

void SetDataPresent(CommChan *Channel)
{
	Channel->DataPresentFlag = TRUE;
}
////////////////////////////////////////////////////////////////////////////
/
// Function: SendToCommModule
//     Programmer: Charles Le
// Input: DataChannel
// Output: None
// Description: This function will set DataPresentFlag to 1
////////////////////////////////////////////////////////////////////////////
/
void SendToCommModule(CommChan *Channel, char *Message)
{
	SetDataPresent(Channel);
   strcpy(Channel->Buff,Message);
}
////////////////////////////////////////////////////////////////////////////
/
// Function: CheckDataPresent
//     Programmer: Charles Le
// Input: DataChannel
// Output: None
// Description: This function will set return DataPresentFlag
////////////////////////////////////////////////////////////////////////////
/

int CheckDataPresent(CommChan *Channel)
{
	return Channel->DataPresentFlag;
}
////////////////////////////////////////////////////////////////////////////
/
// Function: ClearChannelBuffer
//     Programmer: Charles Le
// Input: DataChannel
// Output: None
// Description: This function will clear channel buffer to 0
////////////////////////////////////////////////////////////////////////////
/

void ClearChannelBuffer(CommChan *Channel)
{
	memset(Channel->Buff, 0, sizeof(Channel->Buff));
}
////////////////////////////////////////////////////////////////////////////
/
// Function: InitUdpComm
//     Programmer: Charles Le
```

```
// Input: None
// Output: None
// Description: This function will initialize buffers and set upd port for
//           communication between BL2000 to the remote control.  These ports
will
//           be set up: 4001, 4002, 4003, 4004, 4005
////////////////////////////////////////////////////////////////////////////
/
void InitUdpComm(void)
{
   sock_init();
  #if DEBUGPRINT
  printf("Opening UDP socket\n");
  #endif

   ClearDataPresent(&ManCtrlChan);
   ClearDataPresent(&WayPtChan);
   ClearDataPresent(&GpsChan);
   ClearDataPresent(&CompassChan);
   ClearDataPresent(&ErrorChan);


   ClearChannelBuffer(&ManCtrlChan);
   ClearChannelBuffer(&WayPtChan);
   ClearChannelBuffer(&GpsChan);
   ClearChannelBuffer(&CompassChan);
   ClearChannelBuffer(&ErrorChan);


   sock_mode( &ManCtrlSock, UDP_MODE_NOCHK);

    if(!udp_open(&ManCtrlSock, MAN_CTRL_PORT, /*resolve(REMOTE_IP)*/-1, \
       MAN_CTRL_PORT, NULL)) {
     #if DEBUGPRINT
      printf("udp_open failed!\n");
     #endif
      exit(0);
   }

   sock_mode( &WayPtSock, UDP_MODE_NOCHK);
   if(!udp_open(&WayPtSock, WYPNT_PORT, /*resolve(REMOTE_IP)*/-1, \
       WYPNT_PORT, NULL)) {
     #if DEBUGPRINT
      printf("udp_open failed!\n");
     #endif
      exit(0);
   }

   sock_mode( &GpsSock, UDP_MODE_NOCHK);
   if(!udp_open(&GpsSock,  GPS_PORT,  /*resolve(REMOTE_IP)*/-1,  GPS_PORT,  NULL))
{
     #if DEBUGPRINT
      printf("udp_open failed!\n");
     #endif
      exit(0);
   }

   sock_mode( &CompassSock, UDP_MODE_NOCHK);
   if(!udp_open(&CompassSock, COMPASS_PORT, /*resolve(REMOTE_IP)*/-1, \
       COMPASS_PORT, NULL)) {
     #if DEBUGPRINT
      printf("udp_open failed!\n");
```

```
     #endif
      exit(0);
   }

   sock_mode( &ErrorSock, UDP_MODE_NOCHK);
   if(!udp_open(&ErrorSock, ERROR_PORT, /*resolve(REMOTE_IP)*/-1, \
       ERROR_PORT, NULL)) {
     #if DEBUGPRINT
      printf("udp_open failed!\n");
     #endif
      exit(0);
   }
} //end of function

////////////////////////////////////////////////////////////////////////////
/
// Function: GetMessageFromCommModule
//     Programmer: Charles Le
// Input: CommChan
// Output: return the command from the Remote Controler and clear the present
// flag
// Description: This function will return the command from the Remote Controler
//              and clear the present flag
////////////////////////////////////////////////////////////////////////////
/
char* GetMessageFromCommModule(CommChan *Channel)
{
   ClearDataPresent(Channel);
       return Channel->Buff;
}
////////////////////////////////////////////////////////////////////////////
/
// Function: SendGpsToControlStation
//     Programmer: Charles Le
// Input: None
// Output: None
// Description: This function will send the Gps data to the ctrl station
////////////////////////////////////////////////////////////////////////////
/
void SendGpsToControlStation(void)
{
       tcp_tick(NULL);
       if (CheckDataPresent(&GpsChan))
   {
       send_packet(GPS_PORT,GpsChan.Buff);
      tcp_tick(NULL);
      ClearDataPresent(&GpsChan);
   }
}
////////////////////////////////////////////////////////////////////////////
/
// Function: SendCompassToControlStation
//     Programmer: Charles Le
// Input: None
// Output: None
// Description: This function will send the compass data to the ctrl station
////////////////////////////////////////////////////////////////////////////
/
void SendCompassToControlStation(void)
{
       tcp_tick(NULL);
       if (CheckDataPresent(&CompassChan))
```

```
   {
       send_packet(COMPASS_PORT,CompassChan.Buff);
      tcp_tick(NULL);
       ClearDataPresent(&CompassChan);
   }
}
////////////////////////////////////////////////////////////////////////////
/
// Function: SendErrorToControlStation
//     Programmer: Charles Le
// Input: None
// Output: None
// Description: This function will send the error data to the ctrl station
////////////////////////////////////////////////////////////////////////////
/
void SendErrorToControlStation(void)
{
       tcp_tick(NULL);
   if (CheckDataPresent(&ErrorChan))
   {
       send_packet(ERROR_PORT,ErrorChan.Buff);
       tcp_tick(NULL);
       ClearDataPresent(&ErrorChan);
      }
}

////////////////////////////////////////////////////////////////////////////
//3. IMU
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
/
// Function: getImu
//     Programmer: Kirk Volland
// Input: None
// Output: int 0 success, 1 failure
// Description: This function does RS232 communications with IMU.  Reads in
// chars from serial port until the end of line carriage return detected.
// Parses string for rotation rates.  Saves values to globals.
////////////////////////////////////////////////////////////////////////////
/
int getImu(void)
{
       int i;
   char imuStr[128];
   for(i = 6; i > 0; i--)   //for averaging the rotation rate
   {                                    //shift older data to end of array
       zRotation[i] = zRotation[i-1];   //array element 0 is the newest
       imuTimes[i] = imuTimes[i-1];
   }
   i = 0;
       strcpy(imuStr, ""); //null out any contents
   imuStr[i] = imuChar; //record the leading #
   i++;
   //get an IMU sentence of rotation values, and linear accelerations
   while( (imuChar = serFgetc() ) != -1 && imuChar != '\r' )
       {
       //save chars to array if they are not -1 or the ending
      //rawImuLog[logIndex + i] = imuChar;     //null it out
      imuStr[i] = imuChar;
      i++;
      if( i > 127)
```

```
        i = 0; //reset i so less than max element value of array

   }

   //find the z-rotaion and use ASCII to float to store it's value
   //parse imuStr
   xRot = strtok(imuStr, ",");
   yRot = strtok(NULL, ",");
   zRot = strtok(NULL, ",");
   imuTimes[0] = MS_TIMER;
   if(zRot == NULL)
   {
       zRotation[0] = 999.9;  //error non-value
      return 1;
   }
   else
   {  //subtract 511 from  value
         //conversion 300deg/1024 bits * (2* PI /360 degrees)
       //rotation rate defined  positive  for Counter clockwise body rotation
      //NOTE, nominal 511 value changed to 512 zero reference based on
      // observations of installed IMU when robot stopped.  zRotation in rad/s
     zRotation[0] = (-1* rad( (atof(zRot) - imuZeroPoint) * (float) 300/1024)
);
     return 0;
   }
}

////////////////////////////////////////////////////////////////////////////
//4. compass (I2C)
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: compass
// Programmer: Kirk Volland
// Input: char I2C address and char register to read
// Output: int mag. compass heading 1 to 255
// Description: This function communicates with the digital compass using I2C
// protocol and converts the readout from the compass into int newHeading.
// returns int value.
////////////////////////////////////////////////////////////////////////////
nodebug int compass(char addrsByte, char registerByte)
{
   char cmpd;
   int err;
   int newHeading, i2cTries;
       if (err = i2c_startw_tx() )
   {
       i2c_stop_tx();
      return -10 + err;   // Error.  Return clock stretching too long
   }
  if (err=i2c_wr_wait(addrsByte))
       {
             i2c_stop_tx();
             return -20+err; // Return no ack on slave (retried)
       }
   if (err=i2c_write_char(registerByte))
       {
             i2c_stop_tx();
             return -30+err; // Return no ack on data register 01
       }
       if (err=i2c_startw_tx())
       {
```

```
            i2c_stop_tx();
            return -40+err; // Return too long stretch on read
      }
      if (err=i2c_wr_wait(addrsByte +1))
      {
            i2c_stop_tx();
            return -50+err; // Send read to slave - no ack (retried) return -5
      }

   if( err=i2c_read_char(&cmpd))
   {
            i2c_stop_tx();
                  return -60+err;
   }
   else
   {
                  for ( i2cTries = 0; i2cTries < 4; i2cTries ++)
         {
            if (err = i2c_send_ack())
           {
              // nothing.  try again
           }
           else  // no error.  OK reception of ack from slave
           {
                i2c_stop_tx();
                headingLast = heading;
                heading = (int) (cmpd + MAGVAR);
                            return 0; // normal exit
           }
         }
         i2c_stop_tx();
         headingLast = heading;
         heading = (int) (cmpd + MAGVAR);
            return -70 + err; // not recv'd ack from slave
      }
}


////////////////////////////////////////////////////////////////////////////
//5. Waypoint
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: getwaypoints
// Programmer: Alex, Kirk
// Input: Nil
// Output: Nil
//  Description: This function put the sets of waypoints that was sent from
comms
// module into global data (array of structure for waypoints). NOTE, the GUI's
// polarity is Western hemisphere +.  Code assumes N and W hemispheres.
// Add UPDATED CODE for others LATER.
// Obstacle avoidance has 3 waypoints (index 0,1,2) reserved for nav, but they
// are not used in the WINTER2007 avoidance code.
////////////////////////////////////////////////////////////////////////////
void getwaypoints(void)
{
      char* WayPtStr;
   char  TempLat[20], TempLong[20], TempAction[5];
   int StartIndex;
   char DummyStr[100];
   char tempBuf[20];
```

```
   StartIndex=0;
   strcpy(TempLat,"0.01"); //initialize templat to not 0.0
   memset(TempLat,0,sizeof(TempLat));
   memset(TempLong,0,sizeof(TempLong));
   memset(TempAction,0,sizeof(TempAction));
   WayPtStr=GetMessageFromCommModule(&WayPtChan);
   //WayPtStr=WayPtChan.Buff;
  #if DEBUGPRINT
   printf("WayPtStr=%s\n",WayPtStr);
  #endif


   WayPtStr=WayPtStr+1;
   for (StartIndex= 0; StartIndex<= 12; StartIndex++)
   {
       //Separate waypoint tokens
      if (StartIndex==0)
             strcpy(TempLat,strtok(WayPtStr," "));
      else
         strcpy(TempLat,strtok(NULL," "));

      strcpy(TempLong,strtok(NULL," "));
      strcpy(TempAction,strtok(NULL," "));

      //copy tokens to the WayPoint Structures
      // assume Northern Hemis. lat's
      waypoints[StartIndex].lat= (double) atof(TempLat);
      //assume Western Hemisphere long.
      waypoints[StartIndex].lon= (double) atof(TempLong);
      waypoints[StartIndex].action=TempAction[0];

   #if DEBUGPRINT
       printf("waypoints[%d].lat=%lf\n",StartIndex,waypoints[StartIndex].lat);
       printf("waypoints[%d].lon=%lf\n",StartIndex,waypoints[StartIndex].lon);
       printf("waypoints[%d].action=%c\n",StartIndex,waypoints[StartIndex].actio
n);
   #endif
  // send acknowledgement on Error channel back to laptop
  /*
   sprintf(tempBuf, "waypoint[%d]  Lat:%f  Long:%f\n", StartIndex,\
             waypoints[StartIndex].lat , waypoints[StartIndex].lon);
   strcat(ErrString, tempBuf);
   */
   }
}

////////////////////////////////////////////////////////////////////////////
// Function: makeWayPts
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Fills in array of type Position with the distances in cm North
// and East of the Origin.
////////////////////////////////////////////////////////////////////////////
void makeWayPts(void)
{
       int i;
       double tempLat, tempLong;

       for(i = 1; i < 13; i++)
       {
```

```
      if(waypoints[i].lat != 0)
      {
                  tempLat = waypoints[i].lat;
                  // make radians for math in flat Earth
                  tempLat = (double) rad(tempLat);
       tempLong = waypoints[i].lon;
                  // make radians for math in flat Earth
                  tempLong = (double) rad(tempLong);
                  // record into array of structs using flat Earth approx's
global
         // R1 radius and R2 radius
       // multiply Radii in km by 1E5 to get radii in cm
                  waypointsPos[i].cm_N  =  (long)  R1  *  1E5*  (tempLat  -
initPos_lat);
                  waypointsPos[i].cm_E = (long) R2 * 1E5* cos(initPos_lat) *
\
             (initPos_long- tempLong );
       #if DEBUGPRINT
         printf("tempLat %.8lf initPos_lat %.8lf\n", tempLat, initPos_lat);
         printf("tempLong %.8lf initPos_long %.8lf\n", tempLong, initPos_long);
       printf("waypointsPos[%d].cm_N =%ld\n", i,waypointsPos[i].cm_N);
       printf("waypointsPos[%d].cm_E =%ld\n",i,waypointsPos[i].cm_E);
       #endif
       }
      else
      {
       waypointsPos[i].cm_N = 0;
                  waypointsPos[i].cm_E = 0; //default to Origin if bogus
      }

       }
}
////////////////////////////////////////////////////////////////////////////
//6. Navigation
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: getOriginGPSfromGUI
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Sets the lat and long of the GPSPosition struct for the
// Origin's lat, long from waypoint 0.
////////////////////////////////////////////////////////////////////////////
void getOriginGPSfromGUI(void)
{

       initPos_lat = waypoints[0].lat;  //save the float value as degrees from
GUI
   //toss out decimal portion, keep integer degrees
       OriginPos.lat_degrees = (int) waypoints[0].lat;
       initPos_long = waypoints[0].lon;
       OriginPos.lon_degrees = (int) waypoints[0].lon;
       //make degrees minutes, decimal minutes for GPS style format
   //get the fraction remainder left over in floating pt form
   // and multiply by 60 to make decimal minutes
       OriginPos.lat_minutes  =  (waypoints[0].lat - OriginPos.lat_degrees) *60;
       OriginPos.lon_minutes  =  (waypoints[0].lon - OriginPos.lon_degrees)*60;
       OriginPos.lat_direction = 'N'; //assume N and West hemispheres
       OriginPos.lon_direction = 'W';

}
```

```
////////////////////////////////////////////////////////////////////////
// Function: initNav
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Solves for some global variables R1, R2, and the Origin's
// lat and long in radians to speed up calcs in the flat Earth approx. math.
////////////////////////////////////////////////////////////////////////
void initNav(void)
{
       int i;
       getOriginGPSfromGUI();    // input the GPS position for the Origin
       // set the Origin's GPS lat, long
       //solve for the Origin's lat, long in radians for flat Earth nav approx.

       if(OriginPos.lat_direction=='S') // get the sign convention correct.
              initPos_lat=  0  -  initPos_lat;  //negative  values  in  Southern
hemisphere
       // make radians for math in flat Earth
       initPos_lat = (double) rad(initPos_lat);

   if(OriginPos.lon_direction=='E')
       initPos_long=  0 - initPos_long; //negative values for East hemisphere
       // make radians for math in flat Earth
       initPos_long = (double) rad(initPos_long);
       //find R1 and R2, radii of curvature in km
   //esquared is measure of flatness from WGS84 baseline
       R1 = a*(1 - esquared) / pow( (1- esquared* pow(sin(initPos_lat), 2) ),
1.5);
       R2 = a/(sqrt(1- esquared* pow(sin(initPos_lat),2)));
   DRpos.cm_N = 0;
   DRpos.cm_E = 0; // put the robot at the Origin
   for(i = 0; i< 3; i++) //init the dr, too
   {
       vE[i] = 0;
       vN[i] = 0; // zero initial speed
   }
}

////////////////////////////////////////////////////////////////////////
// Function: makeFakeGPSpos
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Creates a sentence that simulates GGA sentence that the GUI
// expects.  currentLat and currentLong are found from distance N. and dist
// E. of the Origin summed with the lat, long of the Origin. Going Eastward
// subtracts longitude from Origin's longitude.
////////////////////////////////////////////////////////////////////////
nodebug void makeFakeGPSpos(void)
{
       double currentLat, currentLong;
       int latDeg, longDeg;
   float flatDeg, flongDeg;
       float latMin, longMin;

       //find "lat" and "long" from the distance in cm N and E. of the Origin of
flat Earth
   //distN is in cm so divide it by 1E5 for km
       currentLat = DRpos.cm_N / (R1* 1E5) + initPos_lat;
   // lat and long in radians
       currentLong =  initPos_long - DRpos.cm_E / (R2* 1E5 * cos(initPos_lat)) ;
```

```
       //convert to degrees
       flatDeg = deg(currentLat);
       flongDeg = deg(currentLong);

       //break into degrees and decimal minutes
       latDeg =  (int) flatDeg; // get the integer degree values
       longDeg = (int) flongDeg;
       latMin  =  (flatDeg - latDeg)*60;  //get the fraction remainder left over
in

//floating pt form
       longMin =   (flongDeg  -  longDeg)*60; // multiply  by  60  to  make  decimal
minutes
       //fake time?  in fake NMEA sentence
       sprintf(fakeGPSsentence, \
       "GPGGA,223003.8,%02d%09.6f,N,%03d%09.6f,W,1,05,1.1,,\n", latDeg,latMin, \
       longDeg, longMin);
}

////////////////////////////////////////////////////////////////////////////
// Function: navigation
// Programmer: Kirk Volland
// Input: integer last waypoint number, integer next waypoint number for
// array waypointsPos[]
// Output: Nil
// Description: Solves for mag heading needed to move from current position
// (in cm_N and cm_E of Origin) to the next waypoint.
////////////////////////////////////////////////////////////////////////////

nodebug void navigation(int lastWaypt, int nextWaypt)
{

       long dN, dE;     //difference (delta) North centimetrs
                                       //dN = waypoint N_cm - current N_cm
                                       //dE = waypoint E_cm - current E_cm

   double range; //range in  cm to wypt from current position

   float gamma, course;

   int heading_diff;

       //find  delta North (dN) and delta East (dE)
       dN = waypointsPos[nextWaypt].cm_N - DRpos.cm_N;
       dE = waypointsPos[nextWaypt].cm_E - DRpos.cm_E;

       range = sqrt( pow(dN, 2) + pow(dE, 2) ); //pythag. theorem
   theta = rad(300);


}

////////////////////////////////////////////////////////////////////////////
//7. DR
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: wheel
// Programmer: Kirk
// Input: address byte of I2C device, register 0x01, 0x02, or 0x03 memory
//        to read wheel angular speed in Hz.
// Output: integer wheel speed in Hz   or negative integer if I2C error
```

```
// Description: This function uses I2C lib to call device with addrsByte to
// read the value of the memory 0x01 for wheel 1 speed, 0x02 for wheel 2 speed,
// or 0x03 for wheel 3 speed.  Returns wheel speed in Hz.
////////////////////////////////////////////////////////////////////////////
nodebug int wheel(char addrsByte, char registerByte)
{
  char cmpd;
   int err;
   int wheelFrq, i2cTries;
       if (err = i2c_startw_tx() )
   {
       i2c_stop_tx();
      return -10 + err;   // Error.  Return clock stretching too long
   }
  if (err=i2c_wr_wait(addrsByte))
       {
              i2c_stop_tx();
              return -20+err; // Return no ack on slave (retried)
       }
   if (err=i2c_write_char(registerByte))
       {
              i2c_stop_tx();
              return -30+err; // Return no ack on data register 01
       }
       //i2c_Delay(10);
       if (err=i2c_startw_tx())
       {
              i2c_stop_tx();
              return -40+err; // Return too long stretch on read
       }
       if (err=i2c_wr_wait(addrsByte +1))
       {
              i2c_stop_tx();
              return -50+err; // Send read to slave - no ack (retried) return -5
       }

   if( err=i2c_read_char(&cmpd))
   {
              i2c_stop_tx();
                     return -60+err;
   }
   else
   {
                     for ( i2cTries = 0; i2cTries < 4; i2cTries ++)
         {
              if (err = i2c_send_ack())
            {
               // nothing.  try again
            }
            else  // no error.  OK reception of ack from slave
            {
                 i2c_stop_tx();

                 wheelFrq = (int) (cmpd);
                             return wheelFrq; // normal exit
            }
         }
         i2c_stop_tx();

         wheelFrq = (int)(cmpd);  //freq is 2 times the reported
              return wheelFrq; // not recv'd ack from slave
       }
```

```
}

////////////////////////////////////////////////////////////////////////////
// Function: calcVeloRobotFrame
// Programmer: Kirk
// Input: float omega1, omega2, omega3 in rad/s
// Output: Nil
// Description: Finds obsv'd speeds along robot's y and x axis from wheel speed
// data.  Finds direction of observed motion theta WRT to y axis.  Solves
// magnitude of velocity vector and robot chassis rotation.
////////////////////////////////////////////////////////////////////////////
nodebug void calcVeloRobotFrame(float omega1, float omega2, float omega3)
{
       obsVx = RADWHEEL/3 * (-2*omega1 + omega2 + omega3);
   // omega1 perpendicular, no contribution to y axis motion
       obsVy = RADWHEEL/(sqrt(3)) * (omega3 - omega2);
   obsChassisOmega = RADWHEEL/3 * (omega1 + omega2 + omega3);
   // find magnitude of v
   obsV = sqrt(pow(obsVx,2) + pow(obsVy,2));
   // find obsv'd theta (velocity vector WRT robot's y axis)
   if(fabs(obsVx) >0 && fabs(obsVy) > 0)
       obsTheta = atan2(obsVx,obsVy);
   else
       obsTheta = 0;

}
////////////////////////////////////////////////////////////////////////////
// Function: calcVeloEarthFrame
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Takes robot velocity WRT robot's axes
// and translates it into Earth Frame using global var heading.
////////////////////////////////////////////////////////////////////////////
nodebug void calcVeloEarthFrame(void)
{
       float gamma;   // sum of heading and
                                  // observed velocity vector direction obsTheta
       //save 2nd oldest reading as the 3rd oldest
   vE[2] = vE[1];
   vN[2] = vN[1];
   //save newest reading as the 2nd oldest
   vE[1] = vE[0];
   vN[1] = vN[0];

   //update the newest readings in array element 0
   //heading is 0 to 255 1 byte value that corresponds to angle 0 to 2*PI
   gamma =( (float) (2*PI/255 * heading) )+ obsTheta;

   gamma = fmod(gamma, (2*PI)); // don't want values greater than 2*PI
   //test if gamma is in quadrant I, II, III or IV
   if(gamma < PI/2)
   {
       vE[0] = obsV * sin(gamma);   //quadrant I
      vN[0] = obsV * cos(gamma);
   }
   else
   {
       if(gamma < PI)
      {
       vE[0] = obsV * sin(PI - gamma);   // gamma > PI/2 but less than PI
       vN[0] = - obsV * cos(PI - gamma);   // quadrant II
```

```
      }
      else
      {
            if(gamma < 3*PI/2)
         {
             vE[0] = - obsV * sin(gamma- PI);      //quadrant III
             vN[0] = - obsV * cos(gamma- PI);
         }
         else
         {
             vE[0] = - obsV * sin(2*PI - gamma);    //quadrant IV
             vN[0] = obsV * cos(2*PI - gamma);
         }
      }

   }

}

////////////////////////////////////////////////////////////////////////////
// Function: dr
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Takes robot velocity in Earth Frame and multiplies by time
// since last DR to get the estimated position in N. and E components.  Adds
// N. and E. components to last position to get currentDRposition.
////////////////////////////////////////////////////////////////////////////
nodebug void dr(void)
{
       //multipy speed times time in ms since last DR update
   float avgVN, avgVE;
       int i;
   avgVN = 0;
   avgVE = 0;
   //make average velocities over the last three readings
   for(i= 0; i< 3; i++)
   {
       avgVN = vN[i] + avgVN;
      avgVE = vE[i] + avgVE;
   }
   //avgVN = avgVN/3;
   //avgVE = avgVE/3;
   avgVN = vN[0];
   avgVE = vE[0];
       DRpos.cm_N  =  DRpos.cm_N  +  (long)  (avgVN  *  (float)  (MS_TIMER  -
DRtime)/1000);
   DRpos.cm_E = DRpos.cm_E + (long) (avgVE * (float) (MS_TIMER - DRtime)/1000);

   DRtime = MS_TIMER;  //save current time as "start" time for next round of DR
   #if DEBUGDR
   printf("DR posit N %ld cm  E %ld cm\n", DRpos.cm_N, DRpos.cm_E);
   #endif
}

////////////////////////////////////////////////////////////////////////////
//8. Collision Advoidance
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: doRandSpin
// Programmer: Kirk Volland
```

```
// Input: Nil
// Output: long integer value in ms to delay as motors make robot spin
// Description: Calls randNum for value 0 to 1.0 to make a time in milisec
// between 900 and 1900 ms.  Sets globals and returns the time delay back to
// main() loop.
////////////////////////////////////////////////////////////////////////////
long doRandSpin(void)
{
      float randNum;      // a float value for coefficent in range 0.0   <=
randNum < 1.0
      long randTurnTime;  // time in ms.  want values 500 to 1500 ms
   int dir;    // random direction
      randNum = rand();  // find random value from 0 to 1.0
   // random long value from 900 to 900 +1000 = 1500 ms
      randTurnTime = (long) (900 +   randNum *1000);

      dir = (int) randTurnTime % 2;
   chassisOmega = SLOWROTATESPEED;

      spinInPlace();
   return randTurnTime;
}


////////////////////////////////////////////////////////////////////////////
// Function: getIRVolts()
// Programmer:  Kirk Volland
// Input: Nil
// Output: Nil
// Description:  Gets analog voltage inputs from the IR sensors.  Stores values
//  in the irRanges global array.    Copies last round of readings to
irRangesOld1.
// Copies round before the last round to irRangesOld2.
//
////////////////////////////////////////////////////////////////////////////
void getIRVolts(void)
{
      int irNum;
      //record irRangesOld1 to irRangesOld2
      for(irNum = 0; irNum < 6; irNum ++)
      irRangesOld2[irNum] = irRangesOld1[irNum];

      //record last round of IR ranges before overwriting them
      for(irNum = 0; irNum < 6; irNum ++)
      irRangesOld1[irNum] = irRanges[irNum];

      //sample 6 IR sensors
      //start with zero and count up to 5 for A to D on channels 0 to 5
      //newest IR data
      for(irNum = 0; irNum  < 6; irNum++)
      irRanges[irNum] = anaInVolts(irNum, 3);

      //average last three readings
      for(irNum = 0; irNum  < 6; irNum++)
      irRangesAvg[irNum] = (irRanges[irNum] + irRangesOld1[irNum])/2;
}


////////////////////////////////////////////////////////////////////////////
// Function: irCloseContact
// Programmer:  Kirk
// Input: Nil
```

```
// Output: int 0 if no close contact.
//  Return retVal = retVal + 2^(sensor number) if sensor number has close
contact
// If all sensors have closecontact then retVal = 2+ 4+ 8+ 16 + 32 + 64 = 126
// that detected the close in object.
// Description:  Tests irRangesAvg[] for readings that exceed the close range
// limit minus a speed scaling value.  As speed goes up, the IR range
//
////////////////////////////////////////////////////////////////////////////

int irCloseContact(void)
{
       int i;
       int retVal;
       retVal = FALSE;  //default to no close IR contact
       for(i= 1; i < 7; i ++)
       {   //stop if any of the avg two values is close contact

              if(irRangesAvg[i-1]  >= IRSTOP_RNG - .5*v) //scale threshold with
speed
                     retVal = retVal + pow(2,i);
       }
       return retVal;  //return 0 or the integer representation of the sensors
                     //that have close IR contacts
}

////////////////////////////////////////////////////////////////////////////
//9. sonar
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: getSonarRanges(void)
// Programmer: Kirk
// Input: void
// Output: int 0 success -2 fail for no end flag detected  -1 fail no start
flag
// Description: Reads in the start flag then number of
// bytes = NUM_SONAR_AZIMUTHS
////////////////////////////////////////////////////////////////////////////
nodebug int getSonarRanges(void)
{
       int azimuthNum, i;
   char input_char1st, input_char2nd;
   int success; // good test found start of the sonar sentence

   i = 0;  //back up check to prevent getting stuck
                     // waiting in case sonar not operating

   while( (input_char1st = serCgetc()) == -1);
   while( (input_char2nd = serCgetc()) == -1);

   if( input_char1st == 0xFE && input_char2nd != 0xFE)
       success = TRUE;

   // while newest char and 2nd newest char
   // are not equal to start flag followed by some data value that's not 0xFE
       while( !success )
       {

              //eat up chars until we get a match for the start of
              //range sentence flag.  Flag is 0xFE
      input_char1st = input_char2nd;
```

```
            while( (input_char2nd = serCgetc()) == -1);// update the newest
char value
      if( input_char1st == 0xFE && input_char2nd != 0xFE)
            success = TRUE;
   }
   input_char = input_char2nd; // we've recv'd a good data byte, so keep it
       // after we get the start of range sentence flag read in 1 byte
       // for each range value from 0 to NUM_SONAR_AZIMUTHS.  After range bytes
      // get the byte that indicates which value is the newest one of the 20
      // range bytes.  All other range bytes need to be shifted
      // by timeLate (ms) * - robot's velocity vector .
for(azimuthNum = 0; azimuthNum < NUM_SONAR_AZIMUTHS+1; azimuthNum++)
       {
       if(azimuthNum != NUM_SONAR_AZIMUTHS)
            {
            if(input_char == 0x02)
            sonarRanges[azimuthNum] = NOCONTACT; // default no contact range
           else //save float to array
            sonarRanges[azimuthNum]=(float) SONARCONVERSION * input_char;

         }
         else      //save the integer number of the newset reported reading
             newestSonar = (int) input_char;
         while( (input_char = serCgetc()) == -1);   //get another char
       }

      if(input_char != 0xFB)
       return -2;  //error no end flag
      else
       return 0; // normal got full sentence

}

////////////////////////////////////////////////////////////////////////////
// Function: checkSonarRanges()
// Programmer: Kirk
// Input: Nil
// Output: int value of the sectors around robot that have obstructions
// Description: Function tests sonarRanges array for contacts that are closer
// than a minimum value.
//
////////////////////////////////////////////////////////////////////////////
int checkSonarRanges(void)
{

       int retVal;
       int i;
   objectAhead = FALSE; //default to no object detected
       closeContact = FALSE; //default no close in contacts
       retVal = 0;
       for(i = 0; i < NUM_SONAR_AZIMUTHS ; i++)
       {  //test if object ahead of the 300 vector
       if(i >= 15 && i <= 18)
       {
          if(sonarRanges[i] <= MIDRANGE)
          {
              objectAhead = TRUE;
       }
       }

      if(sonarRanges[i] <= MIDRANGE ) //range = 250 means no contacts
             {
```

```
                retVal = retVal + pow(2,i);
          //look for close contacts either side of  300 deg axis
                if(sonarRanges[i] <= CLOSERANGE && (i >= 14 && i <=19))
                      closeContact = TRUE;
            }
      }

      return retVal;
}



////////////////////////////////////////////////////////////////////////////
//10. Motion
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: setMotorVolts
// Programmer: Kirk Volland
// Input: float voltage to motor 1, int direction fwd/rev for motor 1
//        float voltage to motor 2, int dir for motor 2
//        float voltage to motor 3, int dir for motor 3
// Output: int 0 if OK transmission of motor voltages, 1 if failed
// Description: Three float voltages are the analog voltages to output to the
// motors.
//
////////////////////////////////////////////////////////////////////////////
int setMotorVolts(float spd1, int dir1, float spd2, int dir2, float spd3, \
int dir3)
{

      int nIn1, i;
      //do a sanity check on voltages to be sure they are in acceptable range
   // less than or equal 4.0 V or greater than or equal to 0.0
   if( (spd1 <= MAXVOLTS) && (spd2 <= MAXVOLTS) &&(spd3 <= MAXVOLTS) \
      && (spd1 >= STOPVOLTS) && (spd2 >= STOPVOLTS)  && (spd3 >= STOPVOLTS))
      {
            digOut(dir1Chan, dir1);          //set directions 1, 2 and 3
      digOut(dir2Chan, dir2);          //set directions 1, 2 and 3
            digOut(dir3Chan, dir3);
            anaOutVolts(mtr1Chan, spd1); //set voltages for 1, 2 and 3
      anaOutVolts(mtr2Chan, spd2);
            anaOutVolts(mtr3Chan, spd3);
      #if DEBUGPRINT
      printf("setMotorVolts m1 %f  d1 %d  m2 %f  d2 %d  m3 %f  d3 %d\n", \
        mtr1Spd,  dir1,  mtr2Spd, dir2, mtr3Spd,  dir3);
      #endif
            if( (spd1 != STOPVOLTS || spd2 != STOPVOLTS || spd3 != STOPVOLTS )
&& \
      robotNotRotating)
                  robotStopped = FALSE;
            return 0; //normal exit
      }
 }


////////////////////////////////////////////////////////////////////////////
// Function: stopMotors()
// Programmer: Kirk Volland
// Input: nil
// Output: int 0 if OK transmission of motor voltages, 1 if failed
// Description: Sets global direction and motor speed variables to reverse and
```

```
// stop.  Calls setMotorVolts().
//
////////////////////////////////////////////////////////////////////////
int stopMotors()
{
       //dir1 = REV_DIR;
       mtr1Spd = STOPVOLTS;
       //dir3 = REV_DIR;
       mtr3Spd = STOPVOLTS;
       mtr2Spd = STOPVOLTS;
       //dir2 = REV_DIR;

       v = 0; //set globals to zero
       chassisOmega = 0;
        cmdVelocity = 0;

       //set flags
       holdHeading = FALSE;
       robotNotRotating = TRUE;

       //output the voltages
       if(! setMotorVolts( mtr1Spd,  dir1,  mtr2Spd, dir2, mtr3Spd,  dir3) )
              return 0;
       else
              return 1;  // failed to properly set voltages to all motors
}
////////////////////////////////////////////////////////////////////////
// Function: spinInPlace()
// Programmer: Kirk Volland
// Input: nil
// Output: int 0 if OK transmission of motor voltages
//               retVal = retVal + 2^motrNum failed due to topping out
// Description: Uses global chassisOmega for direction and rotational speed.
// Sets v =0 and sets global motor speed vars to SLOWROTATESPEED.
// Calls setMotorVolts().
//
////////////////////////////////////////////////////////////////////////
int spinInPlace(void)
{
       float omega1, omega2, omega3;
   int retVal;
   retVal = 0;        // default expect normal operation, not topping

       if(chassisOmega > 0) // forward direction is counter clockwise rotation
       {
              dir1 = FWD_DIR;
              dir3 = FWD_DIR;
              dir2 = FWD_DIR;
       }
       else
       {
              dir1 = REV_DIR;
              dir3 = REV_DIR;
              dir2 = REV_DIR;
       }
       v = 0;
       cmdVelocity = 0;

       omega1 =    chassisOmega;
   omega1  =  fabs(omega1);  //output  must  be  positive  voltage.  find  absolute
value
       if(omega1 > MAXOMEGA)  //limit output to maximum wheel rotational speed
```

```
      omega1 = MAXOMEGA;
   omega2 =   omega1;
   omega3 =   omega1;
      //set flags
      holdHeading = FALSE;
      robotNotRotating = FALSE;
   mtr1Spd = freqToVolts(omega1);
      if(mtr1Spd > MAXVOLTS)
      {
            mtr1Spd = MAXVOLTS;
            retVal = 2;  //set the 2^1 bit for to show topping out motor1
      }
      mtr2Spd = (.99)* freqToVolts(omega2);  // multiply by coeff. to account

      // for weight distribution
      if(mtr2Spd > MAXVOLTS)
      {
            mtr2Spd = MAXVOLTS;
            retVal = retVal + 4;   //set the 2^2 bit to indicate topping out
motor2
      }
      mtr3Spd = (1.10)* freqToVolts(omega3);   //multiply the voltage for added

        //weight on number 3 wheel
      if(mtr3Spd > MAXVOLTS)
      {
            mtr3Spd = MAXVOLTS;
            retVal = retVal + 8; // set the 2^3 bit to indicate topping motor
3
      }

      //output the voltages
      if(! setMotorVolts( mtr1Spd,  dir1,  mtr2Spd, dir2, mtr3Spd,  dir3) )
            return retVal;
      else
            return retVal;  // failed to properly set voltages to all motors
}

////////////////////////////////////////////////////////////////////////
// Function: vector()
// Programmer: Kirk Volland
// Input: nil
// Output: int 0 if OK
//  Description:  calls  solveMotorSpeeds  to  solve  for  motor  speeds  based  on
global
//  vars  v,  theta  and  chassisOmega.    chassisOmega  negative  is  clockwise
rotation.
// chassisOmega in rad/s
//
////////////////////////////////////////////////////////////////////////
int vector(void)
{
      solveMotorSpeeds();
      //set flags
      holdHeading = TRUE;
      robotStopped = FALSE;    //default value false, then check
      robotNotRotating = TRUE; //default to TRUE , then check

      if(v == 0)
            robotStopped = TRUE;
      if(chassisOmega != 0 && robotStopped) // if chassisOmege is anything
```

```
//BUT zero, then it's rotating
             robotNotRotating = FALSE;  //and robotNotRotating is FALSE

       //output the voltages
       if(! setMotorVolts( mtr1Spd,  dir1,  mtr2Spd, dir2, mtr3Spd,  dir3) )
   {
             return 0;
   }
       else
             return 1;  // failed to properly set voltages to all motors
}


////////////////////////////////////////////////////////////////////////
// Function: solveMotorSpeeds(float chassisOmega)
// Programmer: Kirk Volland
// Input: nil
// Output: int 0 if all calculated motorspeeds are within limits (MAXVOLTS)
//         if motor speed 1 would exceed (topping) then retVal = retVal + 2
//         if motor speed 2 would exceed (topping) then retVal = retVal + 4
//         if motor speed 3 would exceed (topping) then retVal = retVal + 8
// Description: Uses global vars v, theta and chassisOmega.
// Decomposes v into x and y vector components where y axis ponints to motor 1.
// Finds each wheel's angular speed needed to produce the velocity
// vector v in direction theta.  If the calculated analog speed signal exceeds
// MAXVOLTS, the signal is limited to MAXVOLTS.  Global motor speed vars
// mtr1Spd, mtr2Spd, mtr3Spd and global directions dir1, dir2, dir3 are set.
////////////////////////////////////////////////////////////////////////
int solveMotorSpeeds(void)
{
       int retVal;  //return  value  signals  if  voltage  is  within  limits  0  =
normal
                    //     output byte in binary...
                    //     2^3               2^2           2^1           2^0
                    //   topping motor3     topping motor2   topping motor1     0
norm
       char msg[256];
       float vx, vy;
       float omega1, omega2, omega3; //wheel angular speeds in rad /s
       float veloConst;  //scales omega to Hertz measured by optical tachometer
       retVal = 0;  //default OK

       veloConst = 12.566; //12.566 rad/s corresponds to velocity v = 1.0
                                      //scales  velocity  v  to  a  range  zero  to
1.0
       //decompose v and theta into x and y axis components
       vx = v* veloConst * sin(theta);
       vy = v* veloConst * cos(theta);

       //do three dot products to solve for wheel speeds
       //wheel 1
             //       x axis   + y axis + chassis rotation angular speed
       omega1 =    -vx    +    0 +      chassisOmega;

       if(omega1 < 0)
             dir1 = REV_DIR; //reverse if direction is negative
       else
       dir1 = FWD_DIR;

       omega1 = fabs(omega1); //output value  positive voltage. find abs. value
       if(omega1 > MAXOMEGA)  //limit output to maximum wheel rotational speed
```

```
       omega1 = MAXOMEGA;

       //wheel 2
              //       x axis   + y axis
       omega2 =      vx*.5 - vy*.8660 +  chassisOmega; //dot product decomposed
into

                              // x and y components
       if(omega2 < 0)
       dir2 = REV_DIR; //reverse direction
   else
       dir2 = FWD_DIR;

       omega2 = fabs(omega2);
       if(omega2 > MAXOMEGA)  //limit output to maximum wheel rotational speed
       omega2 = MAXOMEGA;
       //wheel 3
              //       x axis   + y axis
       omega3 =     vx*.5 + vy*.8660 + chassisOmega;

       if(omega3 < 0)
                     dir3 = REV_DIR;
   else
       dir3 = FWD_DIR;

       omega3 = fabs(omega3);
       if(omega3 > MAXOMEGA)  //limit output to maximum wheel rotational speed
       omega3 = MAXOMEGA;

       //solve for analog output voltage values
   //apply correction coefficients to motors 2 and 3 voltage signals
   //omega1, omega2, omega3 are the wheel speeds of each wheel in rad/s
       mtr1Spd = freqToVolts(omega1);
       if(mtr1Spd > MAXVOLTS)
       {
              mtr1Spd = MAXVOLTS;
              retVal = 2;  //set the 2^1 bit for to show topping out motor1
       }

   mtr2Spd = 0.99 * freqToVolts(omega2);
       if(mtr2Spd > MAXVOLTS)
       {
              mtr2Spd = MAXVOLTS;
              retVal = retVal + 4;   //set the 2^2 bit to indicate topping out
motor2
       }
       mtr3Spd = (1.10)* freqToVolts(omega3);
       if(mtr3Spd > MAXVOLTS)
       {
              mtr3Spd = MAXVOLTS;
              retVal = retVal + 8; // set the 2^3 bit to indicate topping motor
3
       }
       return retVal;
}

////////////////////////////////////////////////////////////////////////
// Function: freqToVolts(float omegaIn)
// Programmer: Kirk Volland
// Input: float omegaIn radians/s rotation speed wheel
// Output: float voltsOut in Volts to apply to DAC
// Description: Maps desired wheel angular speed to analog signal from DAC.
```

```
////////////////////////////////////////////////////////////////////////////

float freqToVolts(float omegaIn)
{
       float freq; //tachometer frequency (100 cycles per rev )
       float voltsOut; //analog voltage for CPU to send out
       //check if speed is zero or too low
       if (omegaIn < .3770)
              return (0); // minimum rotation rate 6 Hz
       else            //for linear response region of the tach circuit
       {
              freq = omegaIn*100/(2*PI); //tach freq
              voltsOut = (freq + 98.32) / 76.73;
              return voltsOut;
       }
}  //end funct

////////////////////////////////////////////////////////////////////////////
//11. Control
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function: manual_control
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: This function get the command for manual control from comms
// module.  Solves voltages needed to produce the vector direction and velocity
// magnitude.  Then sets analog voltages.
////////////////////////////////////////////////////////////////////////////
void manual_control()
{
       char *buf;//[UDP_BUFF_SIZE];
       char thetaStr[10], veloStr[10], omegaStr[10];


       //strcpy(buf,GetMessageFromCommModule(&ManCtrlChan));
       buf=GetMessageFromCommModule(&ManCtrlChan);

       #if DEBUGPRINT
     printf(" manual buff chan %s\n", ManCtrlChan.Buff);
     printf(" buf : %s\n", buf);
       #endif
       strcpy(thetaStr,strtok(buf," ")); //theta pass in radians
       strcpy(veloStr, strtok(NULL," ")); // velocity should be float 0 to 1.0
       strcpy(omegaStr,strtok(NULL," ")); // omega should be about 0 to 12
   #if DEBUGPRINT
     printf("theta %s, velocity %s chassis omega %s\n", \
       thetaStr,veloStr, omegaStr);
   #endif
       theta = atof(thetaStr);
       v = atof(veloStr);
   cmdVelocity = v;
       chassisOmega = atof(omegaStr);
       if(v ==0 && chassisOmega == 0)
              stopMotors();  //do the easy thing first if just stopping
       else
       {
              if(v ==0 )
                     spinInPlace(); // v is 0 but user commanded rotation
              else
      {
```

```
            cmdHeading =  heading  ;  //convert course to 1 byte integer 0 to
255
         imuHeading = 0;  //reset the imu heading reference
         vector(); //move in that direction
      }

    }
}


////////////////////////////////////////////////////////////////////////////
// Function: initRobot
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Sets default start values for many global vars.  Sets up RS232
// serial ports to IMU and sonar. Inits motors to stopped.  Manual control.
////////////////////////////////////////////////////////////////////////////
void initRobot(void)
{
   //Set up udp ports for communication
       InitUdpComm();
       tcp_tick(NULL);

       //initialize serial port C to sonar detector
   serMode(0);                   //  setup  BL2600  for  three  RS232  three-wire
ports
       serCopen(57140);          //open port with 57140 baud/s
       serCdatabits(0);          //set to 8 bits
       serCparity(0);            //no parity
       serCwrFlush();
       serCrdFlush();
       #if DEBUGPRINT
       printf("serial C init complete\n");
       #endif
   //set up serial port for RS232 from the IMU
   // Open serial port
   serFopen(BAUD232);
   serMode(0);
   // Clear serial buffers
   serFwrFlush();
       serFrdFlush();

       //control (motor) initialization
       dir1 = REV_DIR;
       dir2 = FWD_DIR;
       dir3 = FWD_DIR;
       stopMotors();
       manual_control_flag=1;
       #if DEBUGPRINT
       printf("motors stopped \n");
       #endif
       cmdHeading = 0;
   imuHeading = 0;
   newImu = FALSE;
   imuZeroPoint = 510;
   heading = 0;
   headingLast = 0;
       //init the dr velocities to zero
   obsV = 0;
   obsVx = 0;
   obsVy = 0;
```

```
       //set a default value to somewhere in North America
        //until user sends waypoint data
       waypoints[0].lat = 36.600;
       waypoints[0].lon = 121.870;
       getOriginGPSfromGUI();  //init the nav. Origin until can get GUI input
       initNav();
       //Navigation initialization
       current_wp_count=1;  //start at waypoint 0 and move from there
                                              //to 1, 2, 3, etc.
       newDR= FALSE;
       compass(0xC0, 0x01);
       //error message init
       strcpy(ErrString, "");
}


////////////////////////////////////////////////////////////////////////
// Function: main
// Programmer: Kirk
// Input: Nil
// Output: Nil
// Description: Sets up BL2600 I/O pins.  Calls initRobot and then inits vars
// with scope inside main.
// Enters while loop with costates.
////////////////////////////////////////////////////////////////////////
void main(void)
{
   auto int i,j, k, l, m, sendCompass, channel;
   auto int irSensorVal;
   auto float tempHeading;
       float avgVN, avgVE;
   auto int headingErr;
   auto float imuIntegralHdgErr;

       auto char val[20];
   auto char CompassString[64];//for comms to control
   brdInit();
   // Enable digital outputs DIO 00 and 1 for I2C with external Pullup to +K
   // Jumper J1 set to +K
   // Enable digital output DIO 02,03, 04 using +K 5V with intenal pullups
   digOutConfig(DIGOUTCONFIG);
   for(channel = 0; channel < 2; channel ++)
   {
             digOut(channel, 1); // let SDA and SCL float up
   }
   i2c_init(); // initialize I2C in library function
       anaOutConfig(0,0);  // Analog outputs Unipolar 0 to 10V asyncrhonous
   // Analog inputs
   // Analog Configure channel pairs 0,1,2,3 for Single-Ended unipolar
   // mode of operation.
   // (Max voltage range is 0 - 20v) NOTE BL2600 can do 0 to 20 V
   anaInConfig(0, 0);
       anaInConfig(1, 0);
       anaInConfig(2, 0);
       anaInConfig(3, 0);
   anaOutPwr(1);   // enable power to the DAC
       initRobot();

   i = 0;//init to zero
   j = 0;
   irSensorVal = 0;
```

```
 watchDogTime = MS_TIMER;
 sendCompass = FALSE;
 while(1) //do a lot
 {

     costate
     {
            receive_packet_from_port(MAN_CTRL_PORT);

     if (CheckDataPresent(&ManCtrlChan)==PRESENT)
     {
     #if DEBUGPRINT
     printf("\n ManCtrlChan-> %s\n",ManCtrlChan.Buff);
           #endif
       //ClearDataPresent(&ManCtrlChan);
     }
     waitfor(DelayMs(163));

     }

costate
  {
    receive_packet_from_port(WYPNT_PORT);
    if (CheckDataPresent(&WayPtChan)==PRESENT)
    {
      #if DEBUGPRINT
       printf("\n WayPtSock-> %s\n",WayPtChan.Buff);
      #endif
         //  ClearDataPresent(&WayPtChan);
    }
    waitfor(DelayMs(919));

  }
  costate   //waypt costate
     {
           waitfor(CheckDataPresent(&WayPtChan));
           #if DEBUGPRINT
           printf("man cont flag %d\n",manual_control_flag);
           printf("\n WayPtChan, wypt task-> %s\n",WayPtChan.Buff);
           #endif
           manual_control_flag= FALSE;
           ClearDataPresent(&ManCtrlChan);
           getwaypoints();
    initNav();
           makeWayPts();
    current_wp_count = 1;
    ClearDataPresent(&WayPtChan);
    v = 0.8;
    cmdVelocity = v;
                 theta = rad(300); // set the vector to 060 degrees
                 chassisOmega = 0;

          cmdHeading =  heading  ;  //convert course to 1 byte int 0 to 255
           imuHeading = 0; //reset this to zero for the next leg
                 vector(); //vector robot off in the 060 degree direction

     strcat(ErrString, "Autonomous mode active\n");
    SendToCommModule(&ErrorChan, ErrString);
     SendErrorToControlStation();
           waitfor(DelayMs(2767));
     }
```

```
   //read I2C compass
       //get a compass reading with I2C bus commands
       costate
       {
              if( compass(0xC0, 0x01) == 0)   // call address 0xC0
       newCompass = TRUE;   // set the flag if compass data is OK

      waitfor(DelayMs(331)); // take another reading after delay
       }

       costate //alternate send compass then send GPS to GUI
       {
      if(sendCompass)
      {
              strcpy(CompassString, "");
              tempHeading  =  360  *  (float)  heading/255;   //  1  byte  value  to
degrees
              sprintf(CompassString, "  %f,", tempHeading);

          SendToCommModule(&CompassChan, CompassString);
       SendCompassToControlStation();
         sendCompass = FALSE;
      }
      else
      {
       SendToCommModule(&GpsChan, fakeGPSsentence);
                    SendGpsToControlStation();

         //puts(fakeGPSsentence);
       sendCompass = TRUE;
      }
              waitfor(DelayMs(503)); // take another reading after delay
   }



costate       // get sonar ranges
       {
       serCrdFlush();  //flush out any old data
      waitfor((input_char = serCgetc()) != -1); //wait until data present
                             // there is serial data present on serC
   waitfor(DelayMs(59));  //Some data is present, allow time for PIC to send
                             // about 2 sentences
getSonarRanges();
      waitfor(DelayMs(131)); //repeat after about .157 + 59 = .2 s

       } //end get sonar data costate

       costate     //update motor speeds and random walk away from obstacle
       {
              if (! manual_control_flag )
              {
       checkSonarRanges();
         irSensorVal = irCloseContact();
        if(closeContact ||  irSensorVal >= 32  )
                {
   #if DEBUGPRINT
   printf("stopping for sonar contact\n");
   #endif
   #if DEBUGSONAR
       printf("irSensor Value %d , IR Volts: %.1f %.1f %.1f %.1f %.1f %.1f\n" ,\
   irSensorVal,irRangesAvg[0],irRangesAvg[1],\
```

```
   irRangesAvg[2], irRangesAvg[3], irRangesAvg[4],irRangesAvg[5]);
   #endif

       stopMotors();


       waitfor(DelayMs(211)); //slow down and stop
         //reverse direction for 400 ms
         v = CRAWLSPEED; //assume reverse is clear since just came from there

        theta = rad(120); // set the vector to 120 degrees
        chassisOmega = 0;
        cmdHeading =  heading  ;  //convert course to 1 byte integer 0 to 255
        imuHeading = 0; //reset this to zero for the next leg
        vector(); //vector robot reverse dircection to separate from obstacle
        waitfor(DelayMs(800));

        stopMotors();  //stop after backing up
        waitfor(DelayMs(100));  //wait for motors to stop
        spinDelay = doRandSpin();
        waitfor(DelayMs(spinDelay));
        stopMotors();
        waitfor(DelayMs(500));   //wait and check IR sensors along new route
        if(irSensorVal)   //do this if it was IR detected
        {   if(irSensorVal >= 32)
                    {
                           while(    (irSensorVal    =    irCloseContact())    &&
irSensorVal >=32  )
                                  {
                    // find a random turn duration in ms 900 to 1900 ms
                    spinDelay = doRandSpin();

                    waitfor(DelayMs(spinDelay));
                    stopMotors();
             waitfor(DelayMs(1033));  // delay 1 s to so IR check path
                           // and allow user to send manual commands
                                  }
            }
            else
            {
             waitfor(DelayMs(1033));  //just delay 1 s and check with IR
            }
        }
        else   // do this if it was sonar detected
        {
                    waitfor(DelayMs(2033));   //check the area
                    checkSonarRanges();

                    while(sonarRanges[15]  <  CLOSERANGE  ||  sonarRanges[16]  <
CLOSERANGE \
            ||sonarRanges[17] < CLOSERANGE || sonarRanges[18] < CLOSERANGE  )
            // try a spin in place then wait for sonars to check area clear
             {

             // find a random value of ms between 900 and 1900  turn duration
               spinDelay = doRandSpin();

             waitfor(DelayMs(spinDelay));
               stopMotors();
             waitfor(DelayMs(2033));  // delay 2s.  sonars check path
                           // and allow user to send manual commands
                    checkSonarRanges();
```

```
            }

        } //else

            //closeContact = FALSE;  //kludge fix.  why is it staying TRUE?
            v = CRAWLSPEED; //if clear then go again, but go slow

             theta = rad(300); // set the vector to 060 degrees
             chassisOmega = 0;
            cmdHeading =  heading; //convert course to 1 byte integer 0 to 255
             imuHeading = 0; //reset this to zero for the next leg
             vector(); //vector robot off in the 300 degree direction
            waitfor(DelayMs(600)); //wait to clear obstacle

            for(l = 0; l < 8; l++)
            {
             if(!closeContact)
             {
                    checkSonarRanges();
                    v = CRAWLSPEED;
                   waitfor(DelayMs(600));
              }
              else
                    stopMotors();
                    break;
            }

      } //closecontact


        else   //no close contacts.  Check for medium range contacts
        {
            if(objectAhead)  // then just slow down
            {
                   v = CRAWLSPEED; //update the velocity to slow down
                   vector();
            }

            else   // OK speed back up
      {
               cmdVelocity = 0.8;
                              v = cmdVelocity;
            vector();
              }
       } //else
            }//mancontrol
     waitfor(DelayMs(131));
      } //costate


   costate
   {
       //print sonar ranges every 2 seconds
      #if DEBUGPRINT
       printf("\nranges 0 to 09:  %.1f %.1f %.1f %.1f %.1f %.1f %.1f %.1f %.1f
%.1f \n" ,\
            sonarRanges[0], sonarRanges[1],\
            sonarRanges[2],  sonarRanges[3],  sonarRanges[4], \
            sonarRanges[5],  sonarRanges[6],  sonarRanges[7],   \
            sonarRanges[8],  sonarRanges[9]);
```

```
   printf("\nranges 10 to 19: %.1f %.1f %.1f %.1f %.1f %.1f %.1f %.1f %.1f %.1f
Newest: %d\n" ,\
                  sonarRanges[10],  \
         sonarRanges[11],  sonarRanges[12],  sonarRanges[13], \
         sonarRanges[14],  sonarRanges[15],  sonarRanges[16], \
         sonarRanges[17],  sonarRanges[18],  sonarRanges[19], newestSonar);

      #endif
      waitfor(DelayMs(2000));
   }

       costate
       {
              getIRVolts();
              waitfor(DelayMs(80));  //IR sensors need 40 to 60 ms to update

       }

       costate   // get wheel speeds for dr
       {

       wheel1 = 2 * wheel(0xD0, 0x01);// mult. reported speed by 2 for Hz
         waitfor(DelayMs(1));          // get wheel speed in Hz for each wheel
         wheel2= 2 * wheel(0xD0, 0x02);
         waitfor(DelayMs(1));
         wheel3 = 2 * wheel(0xD0, 0x03);
         obsOmega1 = wheel1 * 2 * PI/100; // make wheel speed in rad/s
         if(dir1 == REV_DIR)
             obsOmega1 = -1 * obsOmega1;
         obsOmega2 = wheel2 * 2 * PI /100;   //negative rotational speeds if
         if(dir2 == REV_DIR)              //wheels  rotating in rev. direction
             obsOmega2 = -1 * obsOmega2;
         obsOmega3 = wheel3 * 2 * PI /100;
         if(dir3 == REV_DIR)
             obsOmega3 = -1 * obsOmega3;
        #if DEBUGDR
             printf("wheel speeds 1,2,3:  %.4f %.4f %.4f \n", \
             obsOmega1, obsOmega2, obsOmega3);
        #endif
         calcVeloRobotFrame(obsOmega1, obsOmega2, obsOmega3);
         newDR = TRUE;  // flag set so dr costate will activate

         // if test the wheels and they're not spinning, robot is stopped
         if( fabs(obsOmega1) < 0.1 && fabs(obsOmega2) < 0.1 && \
             fabs(obsOmega3) < 0.1)
                   robotStopped = TRUE;
         else
             if(robotNotRotating)
                   robotStopped = FALSE;
            waitfor(DelayMs(250)); // wheel sensor updates every 250 ms
       }


   costate  // dead reackon nav
   {
       if(newDR)
      {
         calcVeloEarthFrame();

              dr(); // dead reckon the position
       newDR = FALSE;  // clear the flag 'cuz we've done
      }                  //the DR on  existing data
```

```
}

    costate //do navigation costate
    {
      if(!manual_control_flag && !closeContact)
      {
          navigation(current_wp_count -1, current_wp_count);

      }
      waitfor(DelayMs(2011));  //do navigation every 2 seconds
    }
    costate //fake the GPS for GUI
    {

    makeFakeGPSpos();
    #if DEBUGDR
    printf("\nvE %f vN %f   DR Posit N %ld  DR Posit E %ld\n", \
    vE[0], vN[0] , DRpos.cm_N, DRpos.cm_E);
   #endif
     waitfor(DelayMs(919));
    }


    costate
{
          waitfor( (imuChar = serFgetc() ) != -1 && imuChar == '#');
   waitfor(DelayMs(10));
   if( !getImu())    //update if not error
   {
    newImu = TRUE;
    if(robotStopped)
          zRotation[0] = 0;  //if observe no wheel speed data,
                                          //robot isn't rotating

      if(fabs(zRotation[0]) > IMUROTLIMIT) //ignore rotation if too small
      {
          imuHeading = imuHeading + zRotation[0] * \
          ((float) (imuTimes[0] - imuTimes[1])/1000 );
      }
      else
      {
          zRotation[0] = 0;
      }
    serFrdFlush();

   }
   waitfor(DelayMs(100));

}


    costate  //maintain heading
    {      //do this if want to maintain heading
    if( (holdHeading && (fabs(imuHeading) > IMUHEADLIMIT) \
                              && !robotStopped && robotNotRotating ) )

    {
          //superimpose rotation onto existing velocity vector
          //add or subtract voltage from ALL motors' speeds
          //to spin the robot R or L to counteract any unwanted
          //chassis rotational velocity
```

```
            //solve for integral imuHeading error
            imuIntegralHdgErr = 0;

              #if DEBUGDR
               printf("current Hdg %d  Command Hdg %d\n", heading, cmdHeading);
               #endif
               for(m = 0; m < 6; m++)
               {
                    if(zRotation[m] < 999) //include the value if not error
value
                  {
                          imuIntegralHdgErr = imuIntegralHdgErr +  \
                          zRotation[m] * (imuTimes[m] - imuTimes[m+1]);
                  }
               }

              // use IMU for heading feedback loop
               chassisOmega =  - KAPPA_D * zRotation[0] - KAPPA_P * imuHeading;
               #if DEBUGDR
               printf("zRotation  %.5f \tIMU Hdg %.5f\n", zRotation[0] ,\
                    imuHeading);
               #endif
               vector(); //update the robot's motion vector

             }//end if holdHeading
      newImu = FALSE;
      waitfor(newImu); //do updates about 10 times per sec

      }//end costate


      costate    //manual  control  costate  calls  manual_control()  to  call
control()
      {
        waitfor(CheckDataPresent(&ManCtrlChan));
        manual_control_flag = 1;
        manual_control();
        waitfor(DelayMs(139));
      }
    costate //watchdog for stuck motionless
    {
       if(!manual_control_flag && v != 0 && (obsOmega1< STUCKLIMIT )\
        && (obsOmega2 < STUCKLIMIT) && (obsOmega3 < STUCKLIMIT) )
      {
       //start a timer
         watchDogTime = MS_TIMER;
         waitfor(DelayMs(TIMELIM));
         //recheck after TIMELIM are we still stuck?
         if(v != 0 && (obsOmega1< STUCKLIMIT ) && (obsOmega2 < STUCKLIMIT) \
             && (obsOmega3 < STUCKLIMIT) )
         {

             //if still stuck then do this
                    stopMotors();
             theta = rad( deg(theta)- 180); //go reverse direction
             v = CRAWLSPEED;


             chassisOmega = 0;
            cmdHeading =  heading  ;  //convert course to 1 byte int 0 to 255
             imuHeading = 0; //reset this to zero for the next leg
             vector(); //vector robot off in the 300 degree direction
```

```
                    msDelay(300);  //short duration try to back up
                    stopMotors();

              waitfor(DelayMs(100));  //wait for motors to stop
                    spinDelay = doRandSpin();
                    waitfor(DelayMs(spinDelay));
                    stopMotors();


                    while( (irSensorVal = irCloseContact()) && irSensorVal >=32
)
                           {
              // find a random value of milliseconds between 900 to 1900
                spinDelay = doRandSpin();

              waitfor(DelayMs(spinDelay));
                stopMotors();
       waitfor(DelayMs(1033));// delay about 1s so IR can check path
                                 // and allow user to send manual commands
                           }
            v = CRAWLSPEED; //if clear then go again, but go slow

              theta = rad(300); // set the vector to 060 degrees
              chassisOmega = 0;
            cmdHeading =  heading  ;  //convert course to 1 byte int. 0 to 255
              imuHeading = 0; //reset this to zero for the next leg
              vector(); //vector robot off in the 300 degree direction
            watchDogTime = MS_TIMER;
                    }
       else
              watchDogTime = MS_TIMER; //new time cuz not stuck
      }
      waitfor(DelayMs(5000));  //check every 5 sec
    }//costate
   } //while

} //main
```

LIST OF REFERENCES

[1] Department of Defense (2004, Nov. 15). Photo Archive- U.S. Department of Defense Transformation Official Website. Available: http://www.defenselink.mil/transformation/images/photos/2004-11/Hi-Res/041103-N-4614W-040.jpg, Nov. 2007.

[2] Department of the Air Force (2004, Nov. 15). 'Reaper' moniker given to MQ-9 unmanned aerial vehicle. Available: http://www.af.mil/news/story.asp?id=123027012&page=2, Nov. 2007.

[3] Department of the Air Force (2007, Mar. 14). First MQ-9 Reaper makes its home on the Nevada flightline. Available: http://www.af.mil/shared/media/photodb/photos/070313-F-0782R-115.jpg, Nov. 2007.

[4] Defense Advanced Research Projects Agency (2007, Nov. 4). Overview. Available: http//www.darpa.mil/grandchallenge/overview.asp, Nov. 2007.

[5] Defense Advanced Research Projects Agency (2007, Nov. 4). D2X_1328.jpg 1200x797 pixels. Available: http://www.darpa.mil/grandchallenge/images/photos/11_4_07/D2X_1328.jpg, Nov. 2007.

[6] Department of the Navy (2007). 2007 Engine Cost Message. Available: http://safetycenter.navy.mil/aviation/maintenance/downloads/2007_engine_cost_message.txt, Oct. 2007.

[7] Steber, Dan. (2007, Summer). Fight FOD to Save Lives. *Mech*. [Online]. Available: http://safetycenter.navy.mil/media/mech/issues/summer07/fightfod.htm, Oct. 2007.

[8] Department of the Navy (2002, Nov. 7). Navy NewsStand- Eye on the Fleet- Windows Internet Explorer. Available: http://www.navy.mil/view_single.asp?id=3256, Nov. 2007.

[9] A. Chicoine, "The Naval Postgraduate School's Small Robotics Technology Initiative: Initial Platform Integration and Testing," M.S. thesis. Dept. Physics., Naval Postgraduate School., Monterey, CA, 2001.

[10] B. Miller, "Improvised Explosive Devise Placement Detection from a Semi-autonomous Ground Vehicle," M.S. thesis. Dept. Physics., Naval Postgraduate School., Monterey, CA, 2006.

[11] J. Herkamp, "Deployment of Shaped Charges by a Semi-autonomous Ground Vehicle," M.S. thesis. Dept. Physics., Naval Postgraduate School., Monterey, CA, 2007.

[12] H. Kitagawa, T. Kobayashi, T. Beppu, K. Terashima: "Semi-Autonomous Obstacle Avoidance of Omnidirectional Wheelchair by Joystick Impedance Control," *in Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp 2148-2153, 2001.

[13] O. Purwin, R. D'Andrea. "Cornell Big Red 2003," in: D. Polani, A. Bonarini, B. Browning, K. Yoshida (Eds), Robocup 2003: Robot Soccer World Cup VII, Lecture Notes in Artificial Intelligence, Springer, Berlin 2003.

[14] Wikimedia Foundation, Inc. (2007, Nov. 28). Holonomic-Wikipedia. Available: http://en.wikipedia.org/wiki/Holonomic, Nov. 2007.

[15] S. Dickerson, B. Lapin. "Control of an Omni-directional Robotic Vehicle with Mechanum Wheels," *in IEEE*, pp 0323-0328, 1991.

[16] T. Kalmár-Nagy, P. Ganguly, R. D'Andrea. Real-time trajectory generation for omnidirectional vehicles. in *Proc. 2002 American Control Conf.* Anchorage, AK, 2002, pp 286-291.

[17] R. Rojas, A. Gloye Förster. "Holonomic Control of a robot with an omnidirectional drive," KI- Künstliche Intelligenz, vol. 20, nr. 2, BöttcherIT Verlag, 2006.

[18] Kornylak Corp. (2007). Transwheel Executive Summary. Available: http://www.motionsavers.com/Kornylak/?gclid=CLjOmJDxgpACFRIQYQodqSg5rw, Nov. 2007.

[19] SuperDroid Robots (2006). IG32 24VDC 195 RPM Gear Motor. Available: http://www.superdroidrobots.com/product_info/IG32GMa.gif, Oct. 2007.

[20] *LMD18200 3A, 55V H-Bridge*, National Semiconductor, 2005.

[21] C. Pemma. Pulse Width Modulation. Available: http://www.cpemma.co.uk/pwm.html, Aug. 2007.

[22] Total Micro Customer Support, private communication, July 2007.

[23] Total Micro (2003). PowerStation-100. Available: http://www.total-micro.com/specifications/POWERSTATION100.pdf, Nov. 2007.

[24] Acroname Robotics (2006, Sept. 6). Sonar Ranging Primer. Available: http://www.acroname.com/robotics/info/articles/sonar/sonar.html, Nov. 2007.

[25] A. Pressman, "Fundamental Switching Regulators- Buck, Boost, and Inverter Topologies," in *Switching Power Supply Design*, 2nd ed. New York: McGraw Hill Professional, 1998, pp 3-21.

[26] P. Scherz, "Electronic Circuit Components," in *Practical Electronics for Inventors*, 2nd ed. New York: McGraw Hill, 2007, pp 382.

[27] D. Schelle, J. Castorena (2006, June). Buck-Converter Design Demystified. *Power Electronics Technology* [Online]. Available: http://powerelectronics.com/power_systems/dc_dc_converters/power_buckconverter_design_demystified/, Oct. 2007.

[28] Hyun Il Jun, "Implementation and Testing of a Robotic Arm in an Autonomous Vehicle," M.S. thesis. Dept. Physics., Naval Postgraduate School., Monterey, CA, 2007.

[29] Rabbit Semiconductor (2007). Wolf (BL2600) C-Programmable Single-Board Computer with Ethernet User's Manual. Available: http://www.rabbitsemiconductor.com/documentation/docs/manuals/BL2600/BL2600UM.pdf, Nov. 2007.

[30] *PIC16F631/677/685/687/689/690 Data Sheet*, Microchip Technology Inc., 2007.

[31] GP2Y0A02YK Long Distance Measuring Sensor, SHARP.

[32] J. Borenstein, H.R. Everett, L. Feng, "Sensors for Map-Based Positioning," in *Navigating Mobile Robots*, Wellesley, MA: A K Peters, Ltd., 1996, pp 69-71.

[33] A. Cao, J. Borenstein, "Experimental Characterization of Polaroid Ultrasonic Sensors in Single and Phased Array Configuration," in *Proc. of the UGV Technology Conf. at the 2002 SPIE AeroSense Symp.*, Orlando, FL, pp. 1-5, 2002.

[34] K. Volland, B. Bateman, B. Kerstens, A. Lucrecio, "SE4015 Winter 2007 Class Report on the F.B. Autonomous Vehicle," Mar. 2007, unpublished.

[35] Hobbico, Inc (2007). Futaba S3003 Servo Standard. Available: http://www.gpdealera.com/cgi-bin/wgainf100p.pgm?I=FUTM0031, Nov. 2007.

[36] *Photoreflector P5587, P5588*, Hamamatsu Photonics, 2001.

[37] *Falcon/MX 6DOF Sensor Module Preliminary Quick Start Guide*, O-Navi, LLC, 2004 Dec.

[38] M. Gasperi (2007). PIC NXT Interface. Available: http://www.extremenxt.com/picnxt.html, Aug. 2007.

[39] *The I2C-Bus Specification*, version 2.1, Philips Semiconductors, Jan. 2000.

[40] Devantech Ltd. (2007). CMPS03 documentation. Available: http://www.robot-electronics.co.uk/htm/cmps3tech.htm, Nov. 2007.

[41] C. Le, J. Gamble, Z. Cole, "SE4015 Summer 2006: 'aka Ham Sandwich'," 2006, unpublished.

[42] J. Borenstein, Y. Koren, "Real-time Obstacle Avoidance for Fast Mobile Robots," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 19, no. 5, pp. 1179-1187, Sept./Oct. 1989.

[43] J. Borenstein, Y. Koren, "The Vector Field Histogram- Fast Obstacle Avoidance for Mobile Robots," IEEE Journal of Robotics and Automation, vol. 7, no. 3, pp. 278-288, June 1991.

[44] The MathWorks, Inc. (2007). Video and Image Processing Blockset- Tracking Cars Using Optical Flow Demo. Available: http://www.mathworks.com/products/viprocessing/demos.html?file=/products/demos/shipping/vipblks/viptrafficof.html#1, Nov. 2007.

[45] A. Shimada, S. Yajima, P. Viboonchaicheep, K. Samura, "Mecanum-wheel Vehicle Systems Based on Position Corrective Control," in *Proc. IECON 2005 31st Annu. Conf. of IEEE* pp. 2077-2082.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Richard Harkins
   Department of Physics
   Naval Postgraduate School
   Monterey, California

4. Professor Peter Crooker
   Department of Physics
   Naval Postgraduate School
   Monterey, California

5. Physics Department
   Naval Postgraduate School
   Monterey, California